Roman Dunets

Designing Modern Applications for Virtual Private Cloud

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

11 May 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Roman Dunets Designing Modern Applications for Virtual Private Cloud 50 pages + 3 appendices 11 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Marko Klemetti, CTO Erik Pätynen, Senior Lecturer

Delivering enterprise applications that can operate with high availability, excellent performance, and strong security has always been challenging. Important aspect of delivering application in a distributed environment is application portability and packaging its dependencies. This work attempts to address these challenges using state-of-the-art tools and application delivery methods in a cloud. The goal of the project was to create and set up sustainable and reliable service architecture for existing Software-as-a-Service (SaaS) product Trail in Virtual Private Cloud (VPC).

The basis for all high-performance clouds is a virtualised infrastructure. The focus of the study was Linux containers as it is a modern paradigm in cloud computing. The technology provides efficient and lightweight virtualisation mechanism. Docker platform was used as a toolkit for containerisation. The application architecture was defined in a single YAML file using Docker Compose orchestration tool.

The result of this project is the implemented service architecture for the Trail in VPC using Linux containers allowing to build and run the application within a self-contained execution environment using lightweight container-based virtualisation. The implemented architecture automates the application environment setup and maintenance. The project presents that Linux containers with tools like Docker and Docker Compose can be used to improve the application delivery process within a cloud.

Keywords	Docker, Linux containers, cloud computing, Virtual Private Cloud, multi-container application

Contents

1 Introduction

During the last 15 years, the information technology (IT) industry has been transformed under the influence of the cloud computing technologies. Cloud computing is a new way to provide IT services with Internet-based dynamic and large-scale virtual resources. The backbone behind the idea is Internet-based computing, where shared resources, data and information are provided to computers and other devices on-demand. Cloud computing has such important benefits as lowering costs and higher reliability, scalability and sustainability that traditional computing and hosted services do not have. Currently, modern IT organisations rely on cloud-based technologies and this has caused new application delivery challenges.

1.1 Background

The background of this study is an existing web-based asset management tool Trail, developed and supported by a Finnish company, Trail Systems [1]. This tool took its place in the world of performing arts organisations, helping to manage and track equipment. The Trail product is a modern web-based solution designed to enhance the way customers handle information concerning the equipment usage, movement, purchases and maintenance.

Delivering enterprise applications that can operate with high availability, excellent performance and strong security in a distributed environment has always been challenging. Nowadays the applications are more complex, and they are developing faster than a couple of years ago. Eventually, these trends increase the demands on infrastructure, crew and hardware. Hence, this causes the following challenges to application delivery: slow application deployment methods; high costs of hardware maintenance; slow responding to new business requirements and waste of computing resources. The most important challenges for application delivery within a cloud are application portability and packaging application dependencies.

1.2 Business Challenge

Trail is a modern web product based on Software-as-a-Service (SaaS) model principles, which implies that the vendor or service provider is responsible for hosting the application and making it available to customers through the Internet [2]. Following the contemporary best practices, the customers requirements and well-established stan-

dards of the IT industry, the target of Trail application delivery is a Virtual Private Cloud (VPC) [3]. Application portability and packaging application dependencies are the target challenges for a web application delivery within a cloud. Hence, Trail Systems is looking for a solution to improve its application delivery process to VPC using modern virtualisation technologies.

1.3 Objective of the Study

This paper attempts to address the challenges described in section 1.2 using state-of-the-art tools and methods of application delivery in a cloud. Guided with business challenges, this study intends to find an answer to the following question:

> What are the benefits of utilising Linux containers for providing a long-term lifespan of a SaaS product?

Due to the fact that Linux containers is the new industry standard for providing modern digital services, the objective of this project is to create a service architecture based on Linux containers and adjusted for a VPC.

1.4 Scope and Structure of the Study

The scope of this study is the Trail application delivery process using Linux containers. Hence, the main focus of this study is the Linux containers technology, related open-source projects and common software architectures for provisioning applications using containers. However, the scope of this project excludes details of web applications development and covers only the set of significant decisions about the organisation of the software architecture including the selection of the structural elements and their connections with each other.

The thesis includes seven chapter. Chapter 1 provides the introduction of the project, background of the case company, business problem, objectives, and scope of the project. Chapter 2 contains the study approach, selected tools and technologies. Chapter 3 describes the theoretical framework and provides a literature review. Chapter 4 covers the requirements and methodologies analysis, solution design and implementation. Chapter 5 represents the actual results of the solution through its technical verification and validation. This chapter also discusses the benefits and drawbacks of the solution and its possibilities for further development. Finally, chapter 6 provides a summary of the thesis work and evaluates the results.

2 Research Methods and Materials

This chapter briefly describes the context in which the project was carried out and its workflow organisation. The selected research approach and its design is the main focus of this chapter. Finally, the chapter explains which methods and tools were chosen to achieve the project goal.

2.1 Research Design and Approach

This project aims to improve the application delivery process to VPC using modern virtualisation technologies of the existing Trail product. Therefore, the research process includes all the main stages of a software development life cycle. The research process of the thesis is depicted in Figure 1.

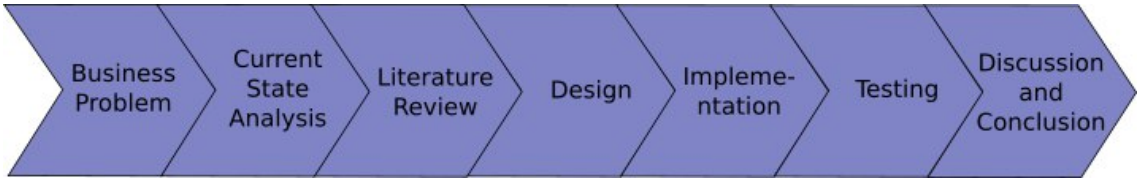

Figure 1. Thesis research design

The project starts with identifying the business problem. The main purpose of this stage is to define the research and development (R&D) project scope, objective and outcome. The current state analysis is the next stage and it is intended to determine the key challenges and requirements of the project. The literature review is about defining a theoretical framework and identification of the best practices to solve the business problem. The following step is the project design which is intended to create a detailed specification of the software solution, accomplishing business goals and technical requirements. The aim of the implementation stage is to build the actual solution for the case product Trail using the chosen tools and technologies. The testing stage is an investigation conducted for the purpose of ensuring that the outcome meets all the business and technical requirements that guided its design and development. The final stage includes discussion about whether the solution is successful and what the next steps are that can be applied to enhance the results.

2.2 Tools and Technologies

This project is based on the Linux Operating System as it has multiple advantages for web application deployment and hosting. Firstly, Linux systems have proven their high

reliability, running for years without failures, eliminating inevitable downtime. Secondly, strong security is the core feature of Linux, making it a reasonable choice for a web server. Thirdly, Linux includes several technologies that allow handling larger workloads while maintaining a consistent level of performance. This is also why the Linux operating system has proven to the most popular choice for hosting web applications. [4]

The main subject of this project is Linux containers (LXC). This technology is a lightweight form of virtualisation which does not require a virtual machine set-up and hardware emulation. This feature uses cgroups resource management facilities and the Portable Operating System Interface for Unix (POSIX) file capabilities for process and network isolation. Each application can separately run within a container being isolated from other process namespaces. Essentially, Linux containers provide the infrastructure to run a complete copy of Linux OS in a container without hypervisors overhead. This approach has many significant advantages described in chapter 3. [5]

Many open-source projects are developing to simplify usage of the Linux containers. Currently the Docker platform is established as an industry standard due to improved management, web front-ends, enhanced visibility in container applications and other advantages. Docker is a container management system that helps to manage and automate containers creation and deployment in an easy and universal way [6]. As a result, the Docker platform was chosen as the most suitable toolkit for building SaaS application architecture in this project.

Distributed applications consist of multiple components (services or applications) that collaborate with each other. In a Docker environment these applications are usually represented as separate containers that are linked together [7]. Docker Compose is an orchestration tool that allows to define a multi-container application with all of its dependencies in a single file, and later spin it up in a single command [7]. The application structure and configuration are held in a single place, which makes applications spinning up simple and repeatable in different environments.

3 Theoretical Background

This chapter provides a brief theoretical background of cloud computing, its connection to virtualisation technologies and best practices in this area. The chapter also introduces Linux containers as a modern paradigm in cloud computing, which provides state-of-the-are lightweight virtualisation. Finally, this chapter is concluded with short discussion on how the LXC technology can change applications development and delivery.

3.1 Cloud Computing

Today, cloud computing covers everything that implies delivering hosted applications and services infrastructure over the Internet. Cloud computing is used in personal life as well as in business life. Gmail, Facebook, YouTube, Twitter and Dropbox, are examples of cloud solutions [8]. The major big enterprises including Amazon, Google, Cisco, Intel, IBM, Novell and Oracle have invested in cloud computing a significant amount of money and provide individuals and businesses with a range of cloud services [8].

Cloud computing is one the ineffable terms which covers web applications and services from different points of view and consequently has many definitions. According to Kris Jamsa (2012), the term describes the abstraction of computers, resources, and services in a web used by software developers and network administrators to implement complex web-based systems [9]. Furthermore, cloud computing is an abstraction of a computing infrastructure with its resources used to provide services over the Internet [10]. Another leading company in cloud computing, Amazon defines it as “the on-demand delivery of IT resources via the Internet with pay-as-you-go pricing” [11]. Despite the diversity of cloud computing definitions, the main idea behind it is to provide computer resources like hardware or software as services hosted and managed at remote data centre.

3.1.1 Cloud Computing Characteristics

The cloud computing model has a set of essential characteristics which enables remote provisioning of IT resources. These characteristics lead to such important features as rapid provisioning, high scalability and cost-efficiency in change management [12]. In addition, they are based on two most important requirements: delivery of IT services on demand and charge based on usage [12]. Another important role of cloud

computing characteristics is to offer cloud computing in an effective manner [13]. Hence, the majority of cloud models include the following characteristics:

- On-demand self-service
- Ubiquitous access
- Resource pooling
- Rapid elasticity
- Measured service
- Multi-tenancy [13].

These characteristics are described in detail in the following chapter.

The on-demand self-service characteristic is based on virtualisation solutions which enable provisioning of computing resources when requested [12]. Every cloud service offers a web interface to Application Programming Interface (API) for customers to provision new servers, Central Processing Unit (CPU), storage and memory [14]. Furthermore, these tools allow users to configure existing servers and reallocate extra resources [14]. Hence, such an infrastructure requires less IT support, saving significant costs on its maintenance.

The ubiquitous access characteristic determines the availability of cloud facilities via the network and accessibility through the APIs provided by the cloud vendor or other standard technologies that are adapted for different client platforms [15]. This means that every cloud customer should be able to connect to and use the platform over the network at any location using any commonly used network technology supported by a cloud vendor [14].

Resource pooling is another important characteristic of cloud computing. Its implementation is based on virtualisation of physical IT resources in the cloud data centres, which allows to pool large-scale computing resources in order to serve multiple cloud tenants [14, 13]. Thus, these resources are dynamically allocated and reallocated according to cloud consumer demands [13]. This was achieved by complete abstraction of the physical layer for both cloud tenants and data centre engineering teams [14].

Elasticity is the ability to expand and shrink computing resources in a cloud without human intervention. Cloud services offer a vast amount of IT resources which can be elastically provisioned and released [12]. Consequently, from the tenant point of view, cloud capabilities available for provisioning often appear to be unlimited and can be utilised in any quantity at any time [15]. Therefore, elasticity is one of those features which makes cloud computing cost-efficient.

The measured service or measured usage characteristic came from the need to charge cloud service customers based on usage. This characteristic involves the capability of a cloud platform to monitor, control, and report resources usage by cloud consumers [13]. Based on these measurements, a cloud vendor is able to charge a customer only for the resources actually used or granted [13]. Furthermore, the measurements are also used by a cloud customer to analyse and optimise the consumption of the resources.

Finally, the last characteristic of a cloud is multi-tenancy. This key characteristic represents the ability of cloud vendors to virtualise their data centres allowing to provide the same physical server for multiple users or tenants [14]. Hence, cloud applications have to support a multi-user environment and consequently guarantee strong isolation and security for each tenant's services, applications and billing mechanisms [12].

3.1.2 Cloud Delivery Models

Cloud services lead commercial and other organisations to extensive benefits and cost savings. Cloud computing aids customers to organise their demands and goals in a coherent manner [14]. As per available information the businesses are focused on running their tasks rather than on costs and return on investment (ROI) [14]. To achieve the maximum profit from a cloud, multiple delivery models have been developed, based on the type of operation and requirements of the business [8]. Figure 2 shows the differences between those models.

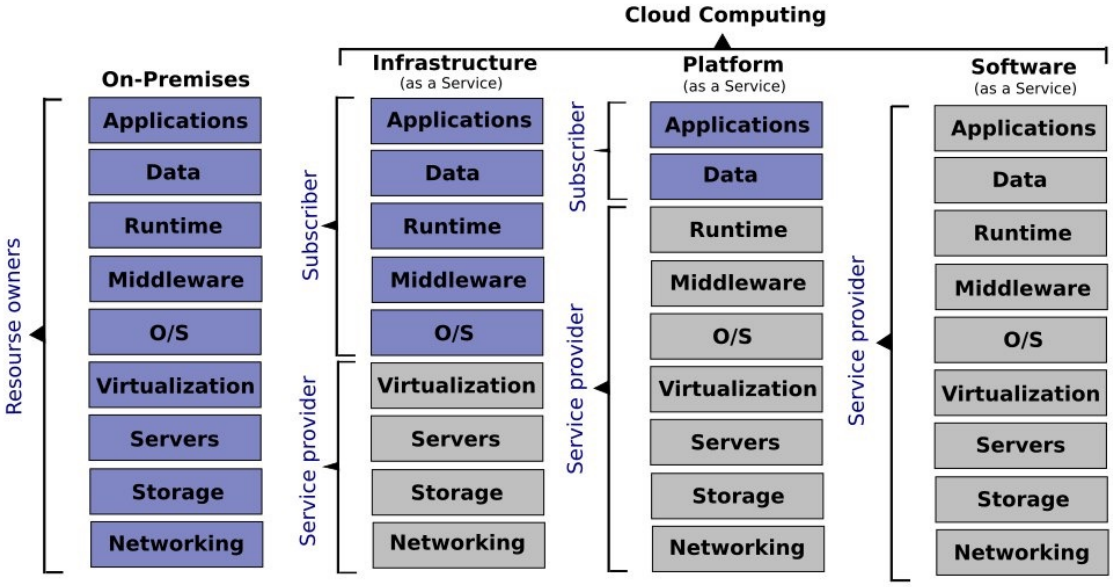

Figure 2. Cloud computing delivery models [2]

There are various types of cloud delivery models but most of them are classified into three categories on the basis of their functionality:

- IaaS (Infrastructure-as-a-Service)
- PaaS (Platform-as-a-Service)
- SaaS (Software-as-a-Service) [13].

The models listed above and their responsibility scopes are demonstrated in Figure 2 and are described more extensively in this section.

Infrastructure-as-a-Service (IaaS) is the first cloud delivery model in Figure 2, providing a basic level of abstraction for the consumer. The IaaS model implies a self-contained IT environment, which consists of infrastructure-centric IT resources and can be accessed and operated through cloud service interfaces and mechanisms [13]. The IaaS compound pattern is based on private and public cloud architectural layers to provide an environment with raw IT resources [16]. The main objective of the IaaS model is to give freedom of control over the environment's configuration and usage to cloud consumers [13]. Figure 2 also shows how responsibilities are divided between the subscriber and provider in the model. As shown in Figure 2, a cloud vendor has full control only over the hardware and virtualisation process (hypervisor) [17]. Hence, the service subscriber has full administration rights in the operating system, middleware, runtime environment, deployed application and the data. To conclude, this model is the most suitable choice when a customer needs raw computing resources, storages and network.

The next model in Figure 2 is Platform-as-a-Service (PaaS). This delivery model is built on top of IaaS and abstracts the most common application stack functions and provides them as a service [2]. Figure 2 displays that in a PaaS model the vendor provides an operating system, middleware and runtime environment to the customer. On the other hand, the subscriber uses services and has full control over the deployed application, data and some aspects of the platform [8]. To achieve this, the PaaS model provides a ready-made environment with installed and configured products and tools to support the whole delivery lifecycle of the customer's applications [13]. Another important aspect to consider is that the model includes the environment of software development, programming languages, compilers and other tools [8]. As a result, PaaS allows geographically distributed development teams to organise a collaborative workflow on software development projects [17]. Finally, it isolates software developers from any complexity of installing tools on workstations at the time of building web applications [17].

Software-as-a-Service (SaaS) is a cloud delivery model designed to provide cloud facilities as a service for the customer. SaaS implementation is similar to the cloud-based service or application, owned and managed by a cloud provider and used on subscription basis by cloud consumers [16]. This model implies that a software product is continuously available, highly scalable, and fault-tolerant [18]. Furthermore, customers do not have any control of the SaaS environment and, consequently, they are not responsible for the client installation, updates or patches [18]. Usually, customers have access only to managing user profiles and configuring application-specific parameters [2]. This implementation is adjusted for multi-tenant usage and, accordingly, the application data, profile, and database are usually hosted separately for each tenant and isolated from others [16]. The service provider controls and manages the underlying infrastructure, business logic of an application, deployment process, and the whole chain of service delivery to cloud customers [2]. To summarise, all of the above mentioned advantages make the Software-as-a-Service delivery model suitable for automating different activities in the business strategy.

3.1.3 Cloud Deployment Models

Cloud is a simplified version of complex, distributed networking systems based on the Internet and Intranet technologies [8]. Most cloud environments have a set of requirements that can be systemised by several general characteristics. These characteristics are used to define cloud deployment models. These models represent different cloud environments, distinguished by ownership, size, and access [13]. In addition, each model defines the relationship of an enterprise network with the Internet, and the degree of a service openness (internal or external clouds) [8]. There are five common types of cloud deployment models:

- Public cloud
- Private cloud
- Virtual Private Cloud (VPC)
- Community cloud
- Hybrid cloud [13].

The models are described below.

The public cloud is the most recognisable cloud deployment model for most customers in the IT industry. This model adopts a virtual environment that shares physical resources and is accessible over a public network, like the Internet, to provide cloud services [19]. The public cloud model provides service through a single infrastructure shared among multiple consumers [19]. Hence, customers share infrastructure costs to

make expenses on infrastructure affordable [12]. On the other hand, due to infrastructure sharing, cloud customisation capabilities are limited. As a result, the cloud provider is responsible for managing and maintaining IT resources that located within its facilities. [20.] The most common examples of public cloud services are email, instant messaging, office and other non-critical applications. To conclude, the public cloud is the most cost-effective cloud deployment model but it is not suitable for workloads with sensitive data or high-level service requirements. [19.]

The private cloud is another type of cloud deployment model, usually used within enterprises for internal needs. This model implies that cloud infrastructure is owned and used by a single organisation [19]. Organisations use private clouds to centralise access to IT resources from different locations, parties and departments [13]. The infrastructure can be managed and maintained by the organisation or a third party, and it can be hosted and deployed on-premises behind a firewall or at a third-party data centre [20, 12]. The most important advantage of the private cloud compared to other cloud deployment models is a high level of customisation and control [20]. To summarise, the private cloud model is a reasonable choice for business-critical applications that work with sensitive data or have high-level service requirements [19].

The Virtual Private Cloud (VPC), also known as a hosted cloud or dedicated cloud, is a variation of public cloud deployment model where a segmented section of the public cloud infrastructure is dedicated to one exclusive customer for private use. The VPC model provides price advantages of using a public cloud vendor with higher security, customisation, storage, networking and other resources [20]. Therefore, the public cloud vendor is responsible for establishing and maintaining the VPC infrastructure [19]. However, resources are not shared with other customers [19]. In addition, the vendor is responsible for securing data transfer to a customer organisation's infrastructure [16]. As a result, the Virtual Private Cloud deployment model is suitable for critical enterprise applications and services [19].

The community cloud is a deployment model intended to provide limited access to a cloud service environment for a specific organisation or community of cloud consumers with shared interests or concerns. Such an infrastructure is usually managed and maintained by multiple organisations, third-party cloud providers that provision a public cloud with limited access or a combination of both [20, 13]. The model is suitable for specific market segments like healthcare, where organisations have common targets, governance, security requirements and policies as designed to provide efficient collaboration [20, 12]. For this reason, each member of the cloud community shares the responsibility for defining and evolving the community cloud [13].

The hybrid cloud is a cloud deployment model based on any combination of private, public, community or VPC models. A common use case for a hybrid cloud model is an organisation maintaining a public cloud service and bursting to it when its private cloud is overstretched and needs extra computing capacity [21]. Furthermore, this model brings infrastructure that can support shared APIs to allow hybrid operations [12]. Consequently, the hybrid cloud provides such benefits as cost-efficiency of the public cloud and high security of the private cloud [12]. However, in some cases managing multiple cloud vendors requires a cloud management system or cloud-broker system [20].

3.2 Linux Containers

In recent years, the IT industry has been transformed with the introduction of a new technology in virtualisation and cloud computing - Linux containers. This is the modern innovative approach which provides a lightweight virtualisation mechanism without emulation of physical hardware [5]. Hence, this fast and lightweight process virtualisation makes the LXC ecosystem a state-of-the-art cloud technology [22]. As a result, the Linux containers technology is radically changing the way software services are developed and used.

The concept of process-level virtualisation is not novel as it was implemented a few years ago in BSD jails and Solaris Zones. Furthermore, the open-source community has been developing other projects intended to enhance process-level virtualisation for a long time. However, these approaches require essential kernel customisation, which is their main drawback. In order to resolve this issue, in recent years multiple projects have been created to provide complete and stable containers virtualisation for the most common Linux kernels. Currently the most popular environments for running multiple Linux containers on a single Linux host are LXC and Docker. Certainly, these solutions have made Linux containers virtualisation an efficient technology for cloud computing. Thereby, more hosting and cloud services businesses are shifting to it. [22]

3.2.1 Virtualisation Basics

The way IT services are provided is dramatically changing by different technologies based on virtualisation. One of the most complete definitions of virtualisation was written by Amit Singh in his work “An Introduction to Virtualisation”: “Virtualisation is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hard-

ware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others" [23]. Another definition states that virtualisation is a process of abstracting physical components into logical objects [24]. In practice virtualisation is a software technology that provides an environment for running multiple operating systems and applications simultaneously on the same machine.

Today, three major types of virtualisation are used: full virtualisation, paravirtualisation and operating system-level virtualisation. As shown in Figure 3, full virtualisation completely separates virtual machines; thus each of them has exclusive access to physical machine resources [25]. In contrast to the full virtualisation approach, paravirtualision provides a supplementary communication channel with an underlying hypervisor for a guest operating system which is used to optimise usage of resources as demonstrated in Figure 4 [22].

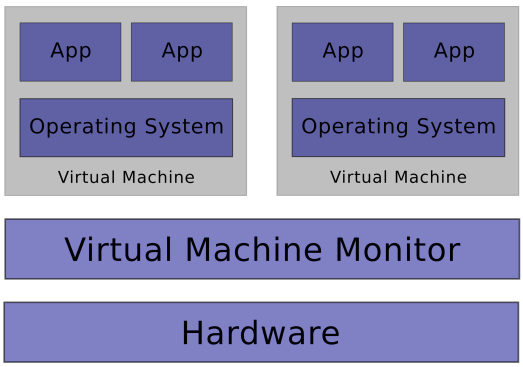

Figure 3. Full virtualisation

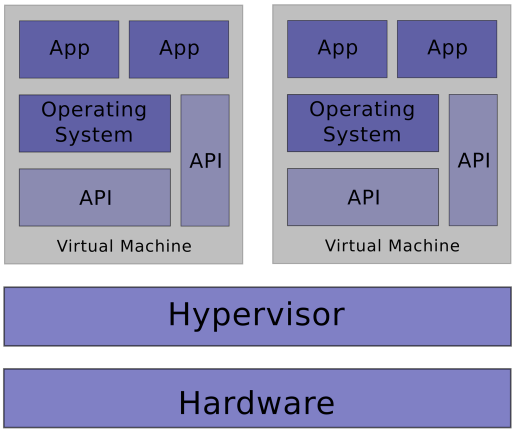

Figure 4. Paravirtualisation

Classic virtualisation systems like VMware or Xen create and run a full operating system on top of a virtualised layer (hypervisor) [26]. These methodologies give exceptionally robust isolation between virtual machines due to the fact that each hosted kernel has a dedicated memory space and defined entry points into the actual hardware [26]. As a result, each of these virtualisation types has their own advantages and disadvantages.

3.2.2 Containers Concept

In 1974 Gerald J. Popek and Robert P. Goldberg described in their article "Formal Requirements for Virtualizable Third Generation Architectures" two types of hypervisors: native or bare-metal hypervisors (type 1) and hosted hypervisors (type 2). Type 1 hypervisors run directly on the host's hardware without an operating system to manage

guest OSs (Figure 5) [24]. Examples of the modern type 1 hypervisors are VMware ESX/ESXi, Microsoft Hyper-V, Citrix XenServer and Oracle VM Server for SPARC. In contrast, type 2 hypervisor is software that runs on top of an operating system (Figure 6) [24]. For instance, VMware Player, VMware Workstation, VirtualBox are type 2 hypervisors.

The Linux containers infrastructure is based on alternative virtualisation methodology. This solution provides virtualisation of the processes in the Linux operating system with less overhead than full virtualisation [27]. In terms of LXC, a container is a process or set of processes running in an isolated environment on a Linux host [22]. Furthermore, this set of processes is isolated from the rest processes running on the machine [28]. Because of this, Linux containers might be encountered with terms such as Virtual Environment (VE) or Virtual Private Server (VPS) [19].

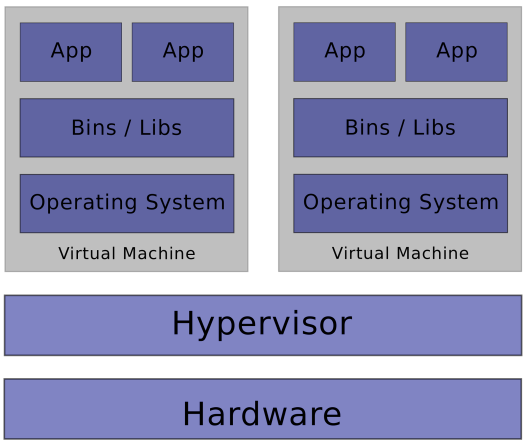

Figure 5. Type 1 hypervisor

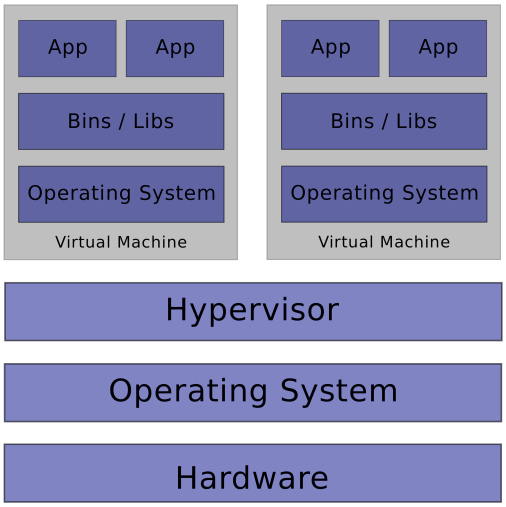

Figure 6. Type 2 hypervisor

Unlike operating system-level virtualisation, full virtualisation or paravirtualisation solutions create separate instances of an operating system or its kernel [22]. Furthermore, hypervisor virtualisation systems use an intermediation layer to run one or more independent virtual machines on a physical hardware [29]. On the contrary, Linux containers run in user space on top of an operating system's kernel [29]. Thus, hypervisors provide a logical isolation at the hardware and virtualisation layer, when containers run in isolation, sharing a single operating system kernel as shown in Figure 5 [28]. This approach extracts the maximum benefit of virtualisation in terms of usage and performance of the resources.

Virtualisation of containers is built around the Linux kernel operating system capabilities. The technology allows to run a full copy of the operating system in a container without an overhead of the running level 2 hypervisor as shown in Figure 7 [5]. Linux containers utilise a single instance of the operating system to run multiple different Linux distributions on the same host [32]. Consequently, container-based virtualisation is also called shared kernel virtualisation, operating system virtualisation or system-level virtualisation [30].

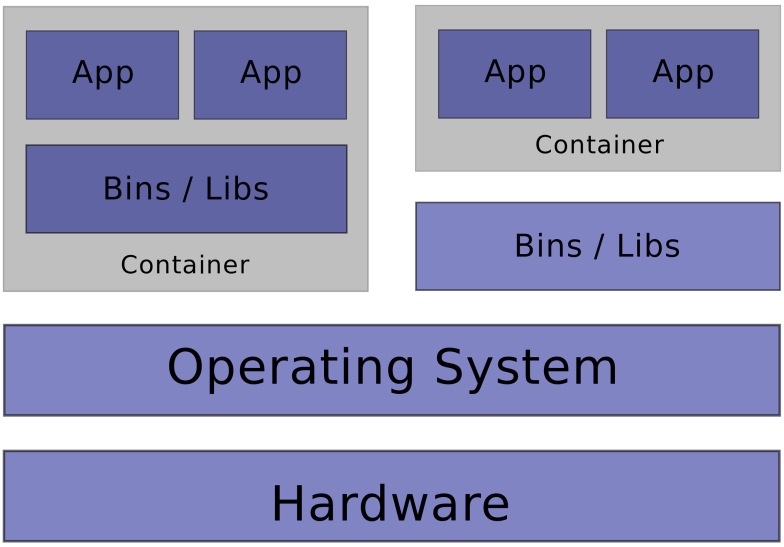

Figure 7. Linux containers virtualisation

Container-based virtualisation is inherent for UNIX-based operating systems. The main idea behind Linux containers is the unique feature of these operating systems to share their kernels with other processes in a system [30]. The mechanism of sharing makes all processes and the file system within the container absolutely visible from the host [5]. On the other hand, due to kernel sharing, the container is limited to the modules and drivers loaded by the host [5]. As a result, this limitation is considered a shared kernel virtualisation drawback.

One of the most important issues in shared kernel virtualisation is isolation of containers. Within Linux containers ecosystem, all containers share the same kernel which implements isolation between them [26]. Furthermore, the kernel supports multiple concurrently running containers [31]. Each of these containers imitate an independent system [31]. In addition, the kernel manages all the global resources so that they are provided separately for each container [31]. Hence, one system is able to accommodate many containers, so that the processes within a container do not have any information about the outside environment or their counterpart containers [31]. Despite the

fact that all containers share the same kernel, each of them has its own Virtualised File System (VFS) and network adapter [27]. Therefore, VFS implies that mounted file systems do not have to be independently tracked for each container [31].

The core feature behind Linux containers is the change root (chroot) operation, which makes shared kernel virtualisation possible. This operation is designed to change the root file system for the current running process in order to isolate it from the rest of the system. This component is also often called a chroot jail and it is a base for container-based virtualisation. Operating system virtualisation provides strong security and isolation for chrooted process or a set of processes by setting up the chrooted system in the way that it has its own isolated root file system. Improvements in the chroot system have made it possible to imitate the whole file system to emulate the entire operating system, creating a VM. [30]

3.2.3 Namespaces and Control Groups

Virtualisation of Linux containers is based on many technologies, which provide sufficient isolation between them in order to make virtualisation possible. One of these key technologies is a namespace isolation concept [27]. This feature allows to avoid running multiple kernels simultaneously on a single physical machine through hypervisors [31]. Instead, namespaces abstract all global resources and accommodate them separately for each container in order to run many systems on a single kernel [31]. The technology separates resources among groups of processes to provide different views of the system to them [32]. This partition allows to isolate members of different containers in the way they do not have any connection with each other and cannot see the entire system [31]. Hence, a group of processes can be encapsulated in a container, being completely independent of processes in other containers [31]. Namespaces are an essential part of Linux containers which enables a lightweight form of virtualisation.

Namespace isolation places an important role in Linux operating system virtualisation. There are six main types of namespaces used in Linux containers for system resources isolation: mount namespace (MNT), UNIX timesharing system (UTS), inter-process communication (IPC), process ID (PID), network and user namespaces [22]. The user namespace enables mapping of user IDs (UID) and group IDs (GID) between a user namespace and the global namespace of a host system [22]. Similarly, the PID namespaces mechanism is another important building block for Linux containers, allowing different processes in different PID namespaces to have the same PID [32]. In addition, PID namespaces allows the process to be check-pointed on one host and be

restored on a different despite the fact that there is a process with the same PID [32]. This feature has the main role in process checkpointing and restoring procedures.

Cgroups is another key technique used in Linux containers. The cgroups project was launched by two developers from Google, Paul Menage and Rohit Seth, in 2006 under the name "process containers" [22]. This subsystem is intended for resource management in the Linux operating system [22]. It enables to limit and isolate the usage of resources for sets of processes. For example, this feature can be utilised to limit the maximum memory process can use through cgroups VFS operations [22]. However, the ecosystem has a resource management solution, which is different from namespaces [32]. As a result, control groups is another essential facility which provides resource management for LXC.

3.2.4 Advantages and Drawbacks of Containers

The Linux containers technology has several advantages compared to hypervisor virtualisation solutions. One of the most important ones is its high performance and resources management efficiency [30]. This is caused by the fact that execution of one isolated function does not require the whole operating system [26]. Shared kernel virtualisation removes one layer of indirection between the actual hardware and isolated task, because the host operating system is sharing its kernel with guests [26]. Hence, this virtualisation methodology provides native performance for all guest systems [30]. This performance beats the most of other common types of virtualisation due to more accurate allocation of resources in the chroot system compared to virtual machines running on a hypervisor [30].

Another important advantage of container virtualisation is the higher density of virtualised systems. The maximum number of chrooted systems closely resembles a standalone system running multiple simultaneous applications [30]. Therefore, in addition native speed performance, system-level virtualisation allows to run more guest systems on a host than standard hypervisor virtualisation solutions [30]. Containers have higher densities due to the significant lightweight of containers in comparison to VMs [22]. In other words, more container instances can be deployed than VMs on the single machine [22]. As a result, Linux containers virtualisation is an efficient and profitable technology for cloud services.

Nowadays, security is a major issue in the IT industry and container-based virtualisation is one of the solutions to many security concerns in cloud computing. The Linux containers technology provides a significant level of security due to the advanced ch-

root system isolation features [30]. Furthermore, containers in Linux are based on tools that enhance process isolation in comparison to the traditional infrastructure of user ids and permissions in UNIX-based operating systems [23]. Introduction of unprivileged containers and architectural re-designing in Linux allowed to restrict containers to their own scopes [26]. Thus, a modern Linux kernel provides strong security features for container virtualisation.

One more crucial profit of using containers is seven times faster software provisioning compared to traditional service hosting [7]. To illustrate this, containers boot and restart in seconds, compared to minutes for virtual machines. These operations take so short a time for containers due to their smaller payloads and absence of hypervisor and guest system overheads. Hence, a faster boot time corresponds to less idle time of service and directly reduces its cost. [28]

Modern cloud computing infrastructures have an issue caused by deployment of services to a cloud. Operating system-level virtualisation is a state-of-the-art solution for this problem of an application's portability. Containers also allow to run a containerised application in different environments like development, test and production [28]. Furthermore, the target platform can be a public cloud, network device, virtual or physical server [28]. Consequently, containers accommodate the ability to move an application across environments and platforms.

Operating system virtualisation in Linux has a few constraints. The major drawback of containers is a compulsory compatibility of guest systems with the host’s kernel due to shared kernel virtualisation. As a result, containers can run only processes compatible with the underlying kernel [26]. This also leads to the fact that Linux containers infrastructure is not able to run Solaris, BSD, OS/X or Windows operating systems within containers unlike hypervisor virtualisation solutions [22]. Finally, the Linux containers technology is the most suitable solution for running Linux applications and servers in a virtual environment.

To summarise, container-based virtualisation has significant advantages over VM-based virtualisation. The most important advantages are the following: native performance, higher density and fast provisioning. In addition, containers can provide easier patching and enhanced security. On the other hand, applications and services running in containers have to be compatible with the underlying kernel. Hence, the ecosystem of Linux containers is a beneficial solution for modern cloud computing companies.

3.2.5 Linux Containers Projects

The numerous projects and tools have been created to simplify the setup and usage of containers. Currently, the most popular containers projects are OpenVZ, LXC, Google containers, Docker and others. These modern technologies transform containers from execution environments to accomplished virtualised hosts [29]. For example, the Docker project utilises novel Linux kernel components like namespaces and control groups [29]. These components provide strong isolation for containers, individual networks and resource management facilities so that multiple containers can exist simultaneously [29]. In addition, container' projects simplify packaging applications with dependencies facilitating their delivery [28]. These projects radically change the way applications are built, shipped, deployed and instantiated [28].

Today, many open-source projects are used to facilitate the deployment and maintenance of containers. These projects provide tools to enable faster application delivery with fewer resources using containers [27]. This allows to deploy containers independently from the host system investing minimum efforts [26]. As an example, nowadays, the container containing a web application, web service or website can be deployed without modifying it on different platforms like Amazon Web Services (AWS), OpenStack and Google Cloud Platform [26]. Furthermore, using container virtualisation gives an extra advantage of having a hybrid infrastructure that utilises services and resources arranged by different providers [26]. Such a hybrid infrastructure can significantly reduce expenses and improve software scalability.

3.2.6 Summary of Containers

In the modern fast-growing world of cloud services, where virtualisation is one of the most important concerns, the Linux containers technology plays a crucial role. In contrast to hypervisors, containers run in isolation and share the same operating system instance [28]. This virtualisation approach improves various aspects of an application delivery by speeding up its deployment, lowering costs, simplifying security and introducing enhanced application designs like microservices architecture and hybrid clouds [28]. Due to these advantages, Linux containers suit most of the modern cloud infrastructure requirements and surpass other virtualisation technologies.

On the other hand, the containers technology is not a universal solution for all cloud virtualisation problems. Despite the fact that containers are designed for efficient deployment and management, they have kernel compatibility limitations [27]. In addition, containers also have performance and security limitations in some particular cases

[27]. Containers virtualisation is a beneficial technology that can provide a maximum profit in many scenarios, but it should be used on a case-by-case basis.

3.3 Docker Project

Docker is an open platform for distributed applications heavily used by developers and system administrators nowadays. This platform encapsulates the Linux containers technology, wraps and extends it with features designed to provide portable images and a friendly user interface [33]. The main idea behind the Docker initiative is to make the containerisation process risk-free, fast and easy [34]. The platform is also a powerful containerisation engine for automating applications packaging, shipping, and deployment [34]. Hence, Docker is a complete solution for creating, maintaining and deploying containers.

3.3.1 Docker Concepts

Introduction of tools like Docker, made Linux containers one of the mainstreams in building cloud infrastructure. Before the introduction of the tool, containers usage involved sophisticated technologies like LXC, which required significant specialist expertise and an amount of manual work [33]. On the contrary, the Docker solution provides a usable and sustainable infrastructure which enables to assemble composite, enterprise-scale, and business-critical applications in a reasonable timeframe [34]. Each of these applications is a lightweight, portable, and self-sufficient container, which can run in different environments [34]. Such an approach allows to use different software components that are distributed as it removes problems with shipping applications to remote locations [34]. Therefore, Docker arranges easy-to-use facilities for code developing, testing and deploying in production as fast as possible.

A Docker platform consists of multiple tools designed to provide smother application delivery within containers. The two most important tools are Docker Engine and Docker Hub. Docker Engine is a fast and convenient interface for creating and running containers [33]. Docker Hub is a repository for distributing Docker images to make them publicly usable, findable and network-accessible [34]. It provides a vast amount of public images, which can be downloaded and used for free. Nevertheless, Docker contains other tools intended to enhance building and deploying complex distributed applications:

- Machine is a tool that automatically provisions Docker hosts and installs the Docker Engine on them.

- Kitematic is a tool that automatically installs Docker, setups processes and provides an intuitive graphical user interface (GUI) for running Docker containers.
- Swarm is a tool that clusters Docker hosts and schedules containers, turning a pool of host machines into a single virtual host.
- Docker Compose is a tool for creating and managing multi-container applications.

The main focus of this project will be on Docker Engine containerisation tool and Docker Compose orchestration tool.

The core Docker technology provides features to support images and containers in order to simplify the management of Linux containers. As a result, images and containers are the main concepts of the Docker project. The Docker image is the basis of the container, which contains changes of the root file system and specified execution parameters for usage within a container runtime [7]. Another important aspect of the Docker project is the container. The Docker container is a runtime instance of a docker image. The container consists of the image, an execution environment and a standard set of instructions [7].

3.3.2 Docker Architecture

Docker has a client-server architecture demonstrated in Figure 8. The architecture contains three major components of the Docker ecosystem: client, host and registry. The client communicates with a daemon in the host designed to build, run, and distribute containers [7]. The client can connect to the daemon located remotely or on the same host [7]. The Docker uses sockets or a RESTful API to provide connection between the client and daemon [35].

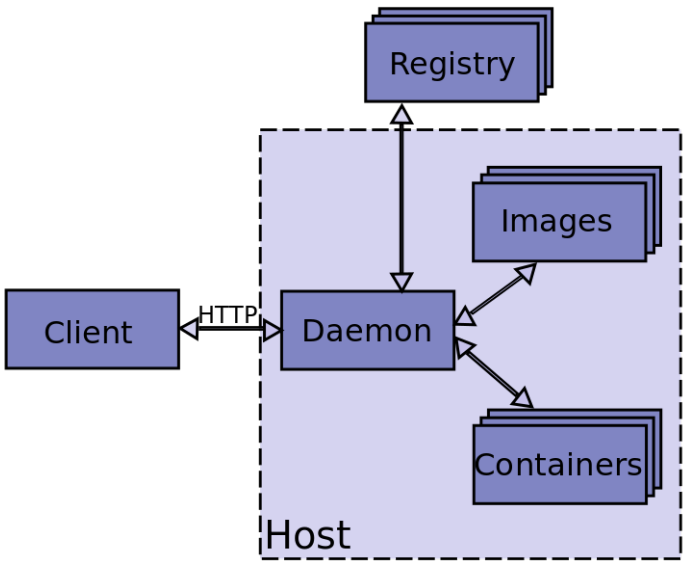

Figure 8. Docker architecture

The host machine is shown in the centre of the scheme in Figure 6. The host runs the Docker daemon and stores images and containers. The daemon cannot be directly used by user, but through the Docker client instead [7]. It provides features to create, manage and run containers, as well as build and store images [36]. Docker daemon is usually launched by the operating system [36].

The Docker client is displayed on the left hand side of Figure 8. It is used as the primary Docker user interface [7]. A user can interact with the client using commands, which will be sent to the daemon and executed. In order to handle communication between the client and the daemon, the Docker infrastructure utilises the HTTP protocol [36]. Due to this, it is easy for other clients to connect with Docker daemon and develop programming language bindings [36]. Furthermore, Docker daemon API is well defined and documented, allowing developers to write software that can directly communicate with the daemon, without using the client [36]. Finally, the Docker client and daemon are distributed as a single binary.

Registries are demonstrated on top of Figure 8. They are public or private stores used for storing and distributing images, where users upload or download them [7]. The most popular registry is provided by the Docker community and it is called Docker Hub. It hosts thousands of public and official images supported by professionals [36]. In addition, many organisations use their own registries to store confidential, commercial or sensitive images as well as avoiding the overhead of downloading images from the Internet [36]. Docker daemon automatically downloads images from registries on pull and run requests if they are not available locally [36]. Hence, Docker registries have an important role as a Docker distribution component.

4 Design and Implementation

This chapter describes how the theoretical knowledge presented in chapter 3 was used in practice to implement a sustainable and reliable service architecture for an existing SaaS product Trail in VPC using Linux containers. This chapter covers the existing Trail application structure and all the phases of the project development: requirements analysis, solution design and implementation. As a result, this chapter demonstrates in practice what kind of benefits the Linux containers technology brings to SaaS products with a long lifespan.

4.1 Trail Product

Trail is a software product that is developed and maintained by a Finnish company Trail Systems. This is a web-based tool for asset and fleet management designed to monitor and maintain items. As a result, Trail has the following business characteristics as a software product:

- All the information in one place - Trail brings together and organises all the information concerning equipment, such as equipment lists, reservations, maintenance and investments [1].
- Barcodes and RFID tags - the software enables to store items with a help of barcodes or RFID tags, allowing to access information concerning items quickly and update it easily [1].
- Maintenance of equipment - the Trail product allows to establish automatic maintenance plans, reducing costly downtime and ensuring that the equipment remains in working condition [1].
- Investment planning assistance - the software collects data that helps to make better investment decisions, avoid unnecessary purchases, lower maintenance costs and save money [1].

The business characteristics listed above describe the high-level features of the product. In order to be available as a web-based service, it has a reliable and portable application architecture, which will be discussed in section 4.2.

Trail application architecture is based on a three-tier software architecture model, which is depicted in Figure 9. The application consists of parts called “tiers”: client tier, application tier and database tier. Each of the parts is deployed on a separate physical machine. The client tier represents the front-end part of the application running in client’s web browser. The application tier is the middle tier, which implements the business logic of the application. Correspondingly to the Model-View-Controller (MVC) pat-

tern, application tier consists of three layers: view, controllers and models. The database tier represents the back-end part of the Trail application which consists of the database server and search engine to persist and fetch information.

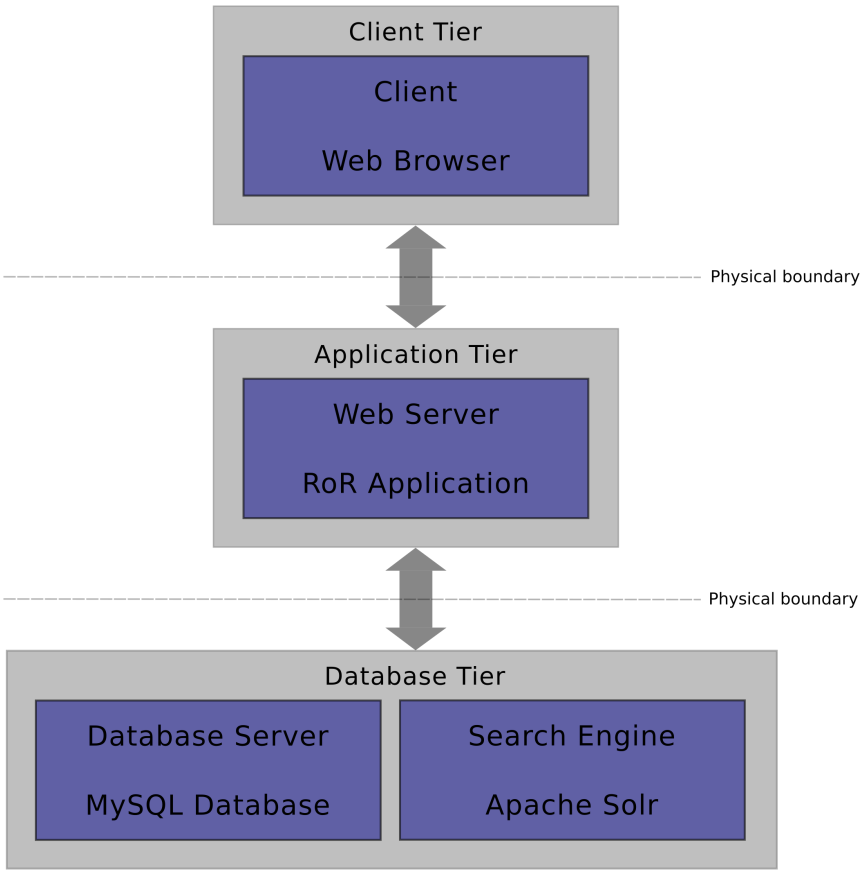

Figure 9. Three-tier software architecture model

The architecture model workflow is based on communication between the front-end and back-end. The workflow process starts when a client sends HTTP requests through a web browser to a web server. The server handles the requests by making queries and updates against the database server. Finally, the query results are passed back to the web server, where an HTML code is generated and returned to the client tier.

The Trail application stack consists of three main parts demonstrated in Figure 9 according to the three-tier architecture model:

- Ruby on Rails application
- MySQL database server
- Apache Solr search engine.

Each of these parts will be discussed in this chapter.

The core of the system is the actual web application based on the Ruby on Rails (RoR) framework. Ruby on Rails is a web application framework written in the Ruby language [43]. The framework is based on the MVC design pattern and it provides default structures for databases, web services, and web pages. The MVC architecture follows the best practices for isolating a business logic from the user interface. As a result, the design pattern provides components in order to keep source code clean for easier development and maintenance. To summarise, the Ruby on Rails framework is intended to facilitate web applications development with less and cleaner code. For more detailed information about Rails, visit the official Ruby on Rails web page [37].

The MySQL database server is the part of Trail's database tier used for data storage and management. MySQL is the most popular open-source Relational Database Management System (RDBMS) in the IT industry [38]. The RDBMD uses Structured Query Language (SQL) as a standard language for accessing a database. However, in accordance to the Ruby on Rails framework, the design of the database has a strong connection to the business logic of an application's model layer. Hence, following the best practices, the Trail web application uses Ruby on Rails migrations in conjunction with the "`rake`" command for the database management. To get more information about MySQL RDBMS, visit the official MySQL web page [39].

Apache Solr is another part of the application stack used for high performance search on the website. Solr is one of the most popular open-source search platforms supported by Apache Lucene project [40]. Solr is highly reliable, scalable and fault-tolerant search engine that provides such features as full text search, near-real-time indexing, replication and load-balanced querying with automised failover and recovery, centralised configuration and other [41]. Furthermore, Solr's search engine has proven its ability to provide high performance search and navigation features for many of the largest websites on the Internet [41]. As a result, it is used in the Trail web application for the same purposes. For more detailed information about Apache Solr, visit the official Solr web page [41].

4.2 Requirements Analysis

The main purpose of this project is to design and implement architecture for the existing software product Trail making it suitable for VPC using lightweight container-based virtualisation. Trail is a large enterprise application, heavily used by thousands of users within their own organisations daily and has an important role in their business activities. Thus, the application architecture have to be adapted for big data manipulations,

extensive parallel processing, utilisation of distributed resources, and complex business logic.

This project also aims to improve the whole Trail application delivery process enhancing the way it is built, shipped and run. As a result, the future application design has a strong accent on the following non-functional requirements:

- Portability - the application will be able to run on a developer's local host, on physical or virtual machines in a remote data centre, or in a VPC without significant changes in a system or application configuration.
- Scalability - the Trail architecture will be adjusted to quickly scale up and teardown components providing efficient IT resources consumption.
- Availability - the software structure will afford maximisation of throughput and minimisation of response time in order to avoid overloading of any single resource.
- Usability - developers will be able to set up and maintain the application architecture infrastructure quickly and easily.
- Reliability - each component of the application stack requires enterprise-grade reliability and security.

Section 4.4 explains how these high-level requirements will be met using modern virtualisation technologies.

This project has also the following technical requirements formed as a result of modern cloud computing trends discussed in chapter 1 and chapter 3:

- Application design is built around Linux containers virtualisation technology.
- Linux containers are used with the help of modern containerisation and orchestration tools.
- The implemented structure is able to run on different environments like developer's local host or production environment in VPC.

Section 4.4 describes an architecture design that suits all of the technical requirements considering non-functional requirements defined in the previous paragraph.

4.3 Architecture Design

This section describes the next stage of the project called architecture design. At this stage the structured solution that meets all of the non-functional and technical requirements described in the section 4.2 is defined and introduced. Furthermore, this section represents how the Trail software product is broken down into structural parts to fit into the architectural solution.

Designing application structure for a long-term lifespan requires thorough selection of reliable technologies, which will be supported and maintained by a company or community during the whole lifespan of the application. After analysis and comparison of existing containerisation tools, Docker platform is selected for the Trail application structure implementation. Docker provides a universal solution to build, package, run, and ship containers using convenient Command Line Interface (CLI) and REST API tools [23]. Consequently, these innovations lowered the barrier to entry point where it became feasible for developers and systems administrators to package applications with dependencies and their runtime environments into self-contained images [23]. In addition, these technologies make containers deployment faster, easier and safer than previous approaches. Another important aspect is the fact that Docker platform brought standardisation to Linux containers. To summarise, Docker is the most sufficient solution in the scope of this project as it completely fulfils the specified requirements.

Designing an enterprise application requires the following best practices to achieve the best outcome from its structure. Thereby, Trail architecture is implemented following the next Docker best practices:

- One application per container - each application or process will run in its own container being isolated from others. Docker is designed to use one application per container in order to make infrastructure management and maintenance easier and more reliable [6].
- Containers are immutable entities - containers are not intended to be modified on runtime but instead restarted from the base image. Therefore, runtime configuration and data are managed outside the containers [42]. For this purpose the Docker volumes are used.
- Official images - official images from Docker Hub are developed and maintained by the projects authoring the software [42]. This makes containers and infrastructure up-to-date with the latest software versions.
- Dockerfile for building custom images - Dockerfile is the preferred method of creating and customising Docker images. It provides a convenient approach for building custom images from scratch, adding instructions that will be executed within the image.

These best practices are applied in the application structure design.

Analysis of the current Trail architecture shows that the software is structured as a distributed application consisting of multiple components which interact with each other. Following the best practices, the application will be fitted into a multi-container structure using modern and convenient containerisation and orchestration tools. Docker Compose is an accepted orchestration tool for Docker containers, used to quickly deploy

complex applications through a single command that calls a human readable configuration file. As a result, Docker Compose will be used as the orchestration tool for the application architecture implementation.

Using tools such as Docker and Docker Compose affords the required level of application's portability, scalability and usability. However, in order to achieve a sufficient level of availability, the infrastructure contains multiple web application and search engine containers in conjunction with load balancers. Finally, each component of the Trail application stack will run in its own container, being isolated from other components, providing a high level of reliability.

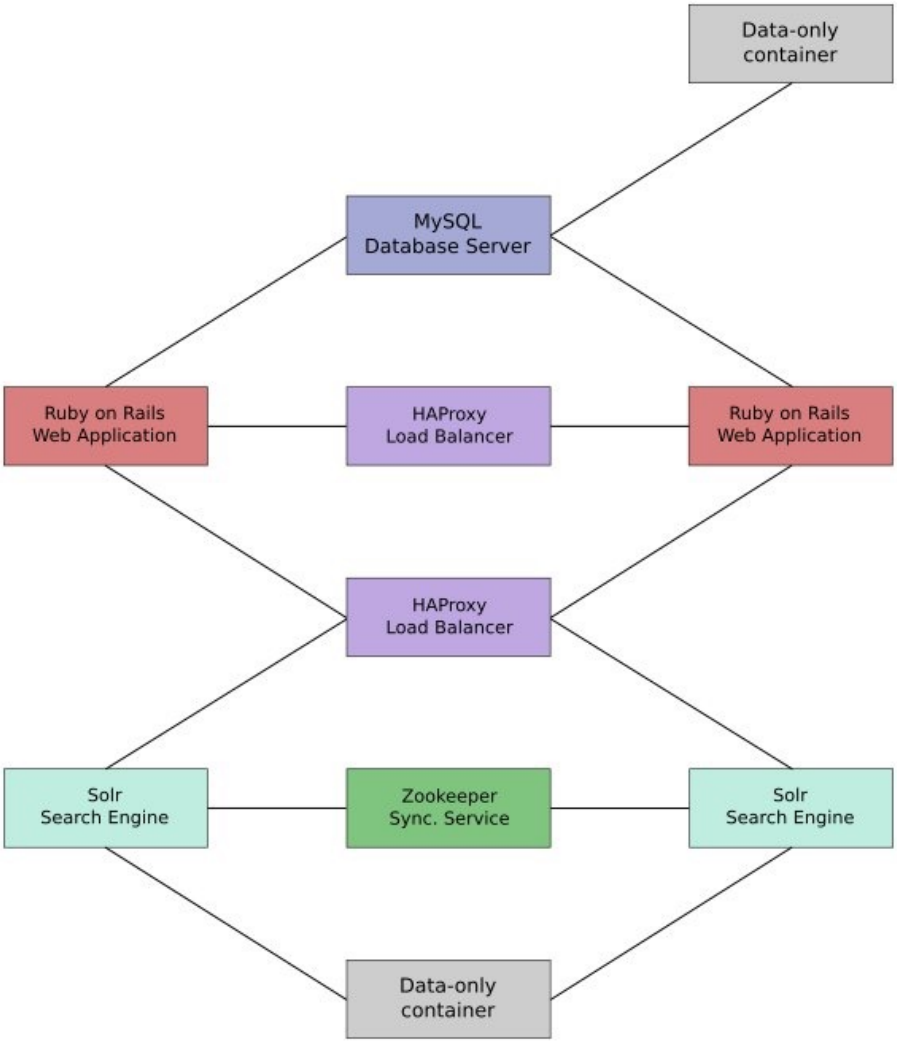

Figure 10. Trail software architecture

Analysis of the requirements and the best practices for Docker and Docker Compose defines a conceptual framework of designing the Trail application architecture. Figure 10 demonstrates how Trail software architecture will be implemented. Each component of the software will run within its own container.

According to Figure 10, the new Trail application architecture will have one instance of the MySQL database server, two instances of Solr search engine and two instances of the Ruby on Rails web application in order to provide a sufficient level of availability. Furthermore, containerisation provides a capacity of extending the architecture flexibly adding more instances of the web application and search engine in the future. Consequently, to distribute the load between those instances, HAProxy load balancers will be introduced [44]. One HAProxy server will be used to manage load for all the web application instances and another will be used to manage load for all the search engine instances. Similarly, to synchronise multiple Solr search engine instances, Zookeeper synchronisation service will be added [45]. Finally, according to the Docker best practices, containers are immutable entities and the runtime data must be stored in dedicated data-only containers. As a result, data-only containers will be added to store data used by MySQL database server and one common data-only container to store data used by Solr search engine instances.

4.4 Prerequisites

4.4.1 Installing Docker on Ubuntu

This section describes how to install Docker Engine on Ubuntu Wily Werewolf 15.10 (64-bit) Linux distribution. From Ubuntu 12.04, Docker is officially supported by the operating system. The following instruction explains how to install Docker from the Ubuntu package repository. The whole installation is performed in the Ubuntu terminal under sudo privileges.

The best practice for installing the Ubuntu package is to resynchronise the package index files from their sources beforehand. This step updates the package lists from the repositories to get information on the newest versions of the packages and their dependencies using the “`apt-get update`” command. The next step is to update all the installed packages in the operating system to the newest version using the “`apt-get -y upgrade`” command.

Within Docker, Advanced Multi Layered Unification Filesystem (AUFS) is used for layering images. In order to enable the AUFS storage driver in Ubuntu, package “`linux-`

`image-extra`" must be installed. The package is recommended to be installed for Ubuntu Trusty, Vivid, and Wily. The "`apt-get -y install`" command is used to install the package.

Afterwards, Advanced Packaging Tool (APT) should be configured to use packages from the new repository [46]. Consequently, the "`apt-key adv --keyserver`" command is used to add Docker repository key to "`apt-key`". Then, the Docker repository is added to APT sources and updated. Finally, Docker Engine can be installed on the system using the "`apt-get -y install`" command. The list of complete commands used to install and configure Docker Engine is provided in Listing 1.

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
$ sudo apt-get -y install linux-image-extra-$(uname -r)
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-key-
servers.net:80 --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
$ echo "deb https://apt.dockerproject.org/repo ubuntu-wily
main" | sudo tee /etc/apt/sources.list.d/docker.list
$ sudo apt-get update
$ sudo apt-get -y install docker-engine
```

Listing 1. Docker Engine installation

Docker daemon binds to a Unix socket, which is owned by the root user by default [35]. Therefore, other users can access it only through "`sudo`" command to run daemon as the root user. In order to avoid using "`sudo`" every time when running Docker commands, a separate group is created called "`docker`" and the target user is added to it (Listing 2).

```
$ sudo usermod -aG docker <user>
```

Listing 2. Creating "docker" group and adding specified user to it

To ensure that the user is running with the correct permissions, the user has to log out and log back. After this, the user should be able to run Docker commands without "`sudo`". On the other hand, an administrator should take into account that "`docker`"

group is equivalent to the root user. Thus, adding users to "`docker`" group has a negative impact on the system security.

After the installation process, Docker Engine is started as a daemon using the "`sudo service`" command. Docker installation is verified by running an official Docker image designed for testing purposes and called "`hello-world`". Command "`docker run`" downloads the image and runs it in a container. If the installation is done successfully and the container run, it prints a message containing text "`Hello from Docker.`". A complete list of commands used to verify Docker installation is shown in Listing 3.

```
$ sudo service docker start
$ sudo docker run hello-world
```

Listing 3. Verification of Docker installation

4.4.2 Installing Docker Compose on Ubuntu

This section briefly describes how to install Docker Composer on Ubuntu Wily Werewolf 15.10 (64-bit) Linux distribution. The administrator should take into account that Docker Compose can be installed only if Docker Engine is installed in the system. The first step to install Docker Compose is to run the "`curl`" command in the Ubuntu terminal as shown in Listing 4.

```
$ curl -L https://github.com/docker/compose/releases/down-
load/1.5.2/docker-compose-`uname -s`-`uname -m` > /usr/lo-
cal/bin/docker-compose
```

Listing 4. Docker Compose installation

If the previous error caused any "Permission denied" error, probably "`/usr/local/bin`" directory is not writable and the administrator needs to install Compose as the superuser. The commands listed in Listing 5 will eliminate the error.

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

Listing 5. "Permission denied" error fix

Finally, the installation can be checked using the “`docker-compose —version`” command. The output of the command is demonstrated in Listing 6.

```
$ docker-compose —version
docker-compose version: 1.5.2
```

Listing 6. Verification of Docker Compose installation

4.5 Implementation

This section goes through the implementation process of the Trail architecture design defined in section 3.4. The main goal of the project was to create a sustainable and reliable multi-container architecture for the existing software product Trail. Hence, this section illustrates how this goal was achieved using tools like Docker and Docker Compose.

Building a multi-container application with Docker Compose includes the following steps:

- Defining the application environment in Dockerfile so that it can run within its own container.
- Defining the services that make up the target application stack in docker-compose.yml (YAML) file so that each of them also runs within its own container.
- Running “`docker-compose build`” command in order to build custom images specified in YAML file to use them afterwards.
- Running “`docker-compose up`” command to start and run the entire application with its services determined in docker-compose.yml file.

The rest of this section covers the steps listed above in more detail.

Following the best practices described in section 3.4, the following Docker images were selected for building the application and its services:

- “`mysql`” - official Docker image for MySQL Community Server
- “`solr`” - official Docker image for Solr search engine
- “`jplock/zookeeper`” - Docker image for Apache Zookeeper (required by official solr image to run multiple Solr nodes in separate containers)
- “`haproxy`" - official Docker image for HAProxy TCP/HTTP Load Balancer
- “`rails`” - official Docker image for the Ruby on Rails environment

All the images listed above are available on Docker Hub and are supported by Docker team. They were used for the Trail architecture implementation and are described later in this section.

The project was developed within a dedicated branch for the multi-container service architecture, which belongs to the Trail GitHub repository. Project directory structure is based on a predefined and standardised Ruby on Rails run-time directory layout. The designed solution was integrated into the existing project structure, minimising architectural inconsistencies and following recommendations from the literature and online guides. The project root directory contains directories and files of the Trail web application. In addition, it contains "`docker`" directory for custom Docker images and configuration files used to build and configure the multi-container architecture. As a result, the directory structure was designed as demonstrated in Figure 11.

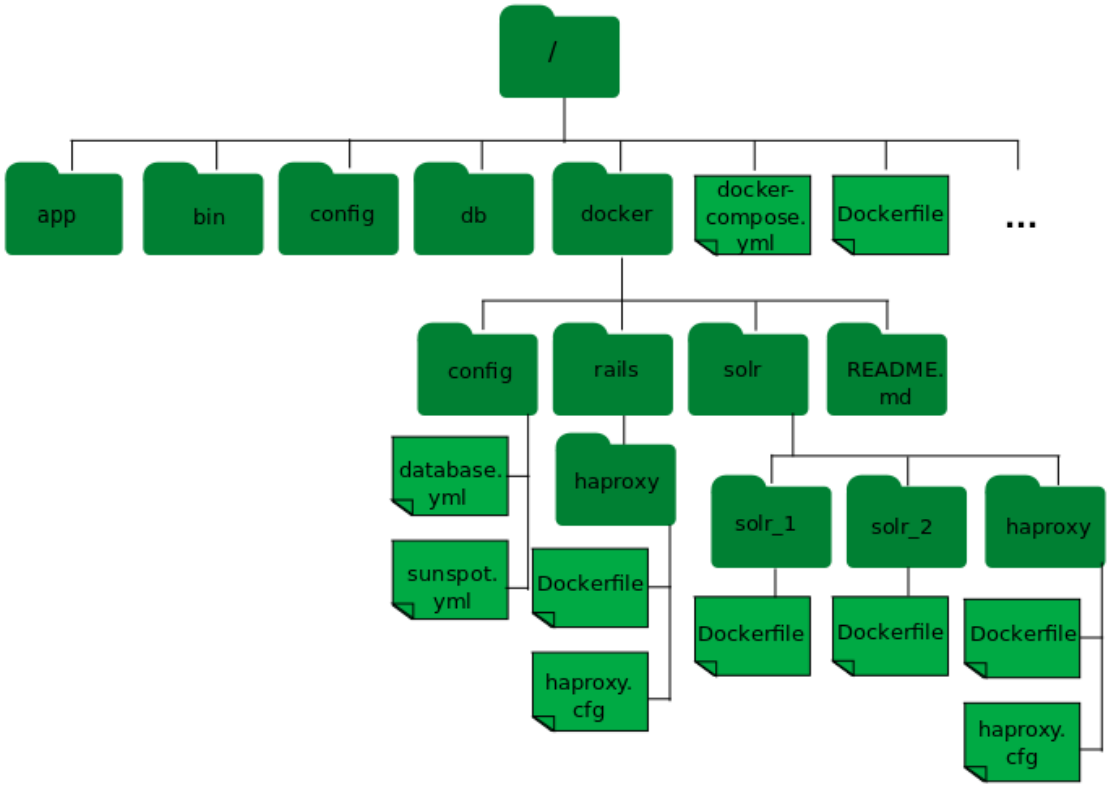

Figure 11. Project directory structure

The first step towards creating the Trail multi-container service architecture was to define an environment for the Trail application so that each component runs within its own container. Docker platform has a tool called Dockerfile that allows to build custom images automatically by reading instructions from the text file [7]. Thereby, Dockerfiles were used to define images for the following services in scope of this project:

- Solr search engine instance (master)
- Solr search engine instance (slave)
- HAProxy load balancer for Solr instances
- Rails web application
- HAProxy load balancer for Rails web application instances.

According to the official “haproxy" image documentation on DockerHub [42], it requires Dockerfile to copy and set up HAProxy load balancer configuration. As a result, both HAProxy load balancers for Rails and Solr services were also defined in Dockerfiles. Implementation details of these custom images are described further in this section.

Dockerfiles for both master and slave Solr instances are shown in Listings 7 and 8 respectively. Both of them utilise “solr” as the base image and contain “MAINTAINER” instruction to determine the author of the Dockerfiles. In addition, both services have “CMD /opt/solr/bin/solr start” command as default command for executing container. The difference between Dockerfiles is that master Solr instance demonstrated in Listing 7 has a command that sets up a default collection for search, which has meaning only in the context of a Solr cluster where a single index is distributed across multiple servers.

```
FROM solr
MAINTAINER Roman Dunets "roman.dunets@trail.fi"
CMD (sleep 10 && /opt/solr/bin/solr create_collection -c
collection1 -shards 2) & /opt/solr/bin/solr start -f -cloud
-z zookeeper:2181
```

Listing 7. Solr Dockerfile (master)

```
FROM solr
MAINTAINER Roman Dunets "roman.dunets@trail.fi"
CMD /opt/solr/bin/solr start -f -cloud -z zookeeper:2181
```

Listing 8. Solr Dockerfile (slave)

Dockerfile instructions for HAProxy load balancer image intended to provide high availability of Solr search engine instances are listed in Listing 9. This custom image employs “haproxy” as the base image and it also determines the author of the Dockerfile. In addition, it has “COPY” command intended to copy HAProxy load balancer configura-

tion file from the current directory to the specified path in the container. The last "CMD" command starts the HAProxy server in the foreground mode using a custom configuration file.

```
FROM haproxy
MAINTAINER Roman Dunets "roman.dunets@trail.fi"
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
CMD haproxy -f /usr/local/etc/haproxy/haproxy.cfg
```

Listing 9. HAProxy load balancer Dockerfile for Solr nodes

According to the designed solution, two instances of the Solr search engine were added to the application architecture. In this case, the Rails web application is able to write to the master node and read from either master or slave Solr nodes. Therefore, when the web application sends an update request, HAProxy directs it to the master instance. On the other hand, when the web application sends a read request, HAProxy balances it between both nodes. To set up this infrastructure for Solr search engines, "solr/haproxy/haproxy.cfg" configuration file was used (Listing 10). The complete content of the configuration file is provided in Appendix 1.

```
frontend solr_lb
    bind 0.0.0.0:8080
    acl master_methods method POST DELETE PUT
    use_backend master_backend if master_methods
    default_backend read_backends

backend master_backend
    server solr-a solr_1:8983 weight 1 maxconn 512 check

backend slave_backend
    server solr-b solr_2:8983 weight 1 maxconn 512 check

backend read_backends
    server solr-a solr_1:8983 weight 1 maxconn 512 check
    server solr-b solr_2:8983 weight 1 maxconn 512 check
```

Listing 10. HAProxy load balancer configuration file for Solr nodes

Dockerfile for the Ruby on Rails web application custom image is demonstrated in Listing 11. This custom image employs “rails” as the base image with “onbuild” tag and it also determines the author of the Dockerfile. Furthermore, the file has “COPY” instruction intended to copy “database.yml” and “sunspot.yml” Rails configuration files to the container. The last “CMD” instruction configures the image to create the database, run Ruby on Rails migrations, run “db/seed.rb” file and start Rails server when the container sets up.

```
FROM rails:onbuild
MAINTAINER Roman Dunets “roman.dunets@trail.fi"
COPY docker/config/database.yml /usr/src/app/config
COPY docker/config/sunspot.yml /usr/src/app/config
CMD rake db:create db:migrate db:seed && rails server
```

Listing 11. Ruby on Rails web application Dockerfile

The Trail architecture design has a HAProxy load balancer to provide high availability of the Ruby on Rails web application instances. The HAProxy server balances the traffic for the Trail application running on multiple web servers by distributing requests across them. Listing 12 shows the content of a Dockerfile located under “rails” directory and intended to build the custom load balancer image.

```
FROM haproxy
MAINTAINER Roman Dunets "roman.dunets@trail.fi"
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
CMD haproxy -f /usr/local/etc/haproxy/haproxy.cfg
```

Listing 12. HAProxy load balancer Dockerfile for Rails web application nodes

Listing 12 contains the same instructions as Listing 9, which includes copying HAProxy load balancer configuration file from the current directory to the specified path in container. The part of the configuration file is represented in Listing 13. The listing contains configuration for both the front-end and back-end settings of the load balancer. In this case, HAProxy is configured to listen incoming requests on port 80. The complete content of the configuration file is provided in Appendix 2.

```
listen webfarm 0.0.0.0:80
    mode     http
```

```
stats    enable
stats    uri /haproxy?stats
balance  roundrobin
option   httpclose
option   forwardfor
server   webserver-a solr_1:80 check
server   webserver-b solr_2:80 check
```

Listing 13. HAProxy load balancer configuration file for Rails web application nodes

Docker Compose provides a convenient way of defining the entire multi-container application in a single configuration file, which is used to spin up the application using one command. Hence, the following step was to define the multi-container application configuration within a "`docker-compose.yml`" file. This is the YAML file used to define services, networks and volumes for the application architecture. In addition, it contains service definitions with settings which are applied to each running container [7].

YAML configuration file defines components for the multi-container application architecture depicted in Figure 10 in section 3.4. Therefore, the configuration sets up the following services:

- MySQL database server
- Data-only container for MySQL database server
- Two instances of Solr search engine as separate services
- Zookeeper synchronisation service
- Data-only container for both instances of Solr search engine
- HAProxy load balancer for both instances of Solr search engine
- Two instances of Ruby on Rails web application (Trail)
- HAProxy load balancer for both instances of Rails web application (Trail).

The rest of this section describes and explains how each of these services are configured using Dockerfiles and "`docker-compose.yml`" file.

The first of the YAML file is represented in Listing 14 and it is used to set up MySQL database server container. The container has name "`db`" and it employs "`mysql`" as the base image. According to the architecture design, the data must be stored outside of the service container in a data-only container. Hence, service mounts the volume from data-only container called "`db_data`". Furthermore, the container has defined the environment variable to set up MySQL database server root password to "`password`". However, such a weak password was used in this thesis only for demonstration pur-

poses. In a production environment all the passwords should follow the company' policy rules. Finally, the service maps port 3306 in the container to port 3306 on the host machine. This setting allows to connect to the MySQL database server instance running in the container from the host machine.

```
db:
  image: mysql
  volumes_from:
    - db_data
  environment:
    - MYSQL_ROOT_PASSWORD=password
  ports:
    - 3306:3306
```

Listing 14. MySQL database server container configuration in YAML file

The next part of the configuration in "`docker-compose.yml`" file is shown in Listing 15 and it is intended to set up data-only container for MySQL database server. This container has name "`db_data`" and it uses "`mysql`" as the base image. Due to the fact that this container is used only for data storage, it reuses the same image used for MySQL database server container so that all containers are using layers in common, saving disk space. In Listing 14, "`db`" service mounts the "`/var/lib/mysql`" volume using "`volumes_from`" setting. As a consequence, "`mysql`" directory located under "`/var/lib/`" directory in "`db`" container is mapped to the same directory in "`db_data`" container, allowing to persist the data necessary to be shared between the containers. Finally, "`mysql`" image runs the command in purpose to start MySQL database server by default when a container is created. However, this container is intended only to store the data. As a result, "`entrypoint`" option is used to override default command with "`/bin/true`" command. This command returns 0 and, consequently, immediately stops the container.

```
db_data:
  image: mysql
  volumes:
    - /var/lib/mysql
  entrypoint: /bin/true
```

Listing 15. Data-only container configuration for MySQL database server in YAML file

According to the designed solution, two containers for Solr search engine with names “solr_1" and “solr_2" were added to the application architecture as demonstrated in Listing 16. Both containers are based on the custom images, which Dockerfiles are located in “solr" directory under the project root directory. In addition, Solr is bundled with Zookeeper and uses it as a repository for cluster configuration and coordination. Therefore, links to Zookeeper synchronisation service are added to both Solr containers. Finally, port 8983 is mapped for “solr_1" and “solr_2” containers on the host machine to ports 8984 and 8985 respectively.

```
solr_1:
  build: solr/solr_1
  links:
    - zookeeper
  ports:
    - 8984:8983

solr_2:
  image: solr/solr_2
  links:
    - zookeeper
  ports:
    - 8985:8983
```

Listing 16. Solr search engine containers configuration in YAML file

The following part of the YAML file configuration is listed in Listing 17 and it is employed to set up Zookeeper synchronisation service for both instances of Solr search engine. The service has name “zookeeper” and it uses “jplock/zookeeper” as the base image in according to the official Solr image documentation on Docker Hub. In addition, the port mapping option is used to expose “zookeeper” container port 2181 on the host machine.

```
zookeeper:
  image: jplock/zookeeper
  ports:
    - 2181:2181
```

Listing 17. Zookeeper synchronisation service container configuration in YAML file

In accordance with the designed Trail architecture, the HAProxy load balancer is used to distribute incoming search queries between the Solr search engine instances. The Docker Compose configuration for the HAProxy server is shown in Listing 18. The load balancer container uses a custom image located in "`solr/haproxy`" directory under the project root directory. It has the name "`solr_lb`", two links to both Solr containers and exposes port 8080 on the host machine.

```
solr_lb:
  build: solr/haproxy
  links:
    - solr_1
    - solr_2
  ports:
    - 8080:8080
```

Listing 18. HAProxy load balancer configuration for Solr instances in YAML file

The next part of the YAML file is demonstrated in Listing 19 and it sets up a data-only container for both Solr instances. The container has the name "`solr_data`" and it reuses the same image used for Solr containers, saving disk space and network resources. Furthermore, the service mounts the "`/var/lib/mysql`" volume using "`volumes_from`" setting to persist data so that it can be shared between Solr containers. Finally, following the same pattern as "`db_data`" container, "`solr_data`" has "`bin/true`" command as an "`entrypoint`" option.

```
solr_data:
  image: solr
  volumes:
    - /opt/solr/server/solr
  entrypoint: /bin/true
```

Listing 19. Data-only container configuration for Solr instances in YAML file

Ruby on Rails web application configuration part in "`docker-compose.yml`" file is shown in Listing 20. In according to the initial design, two instances of the Trail web application are created with names "`rails_1`" and "`rails_2`", respectively. Both containers use a custom image, which Dockefile is located in "`rails`" directory under

project root directory. In addition, both images have links to the MySQL database server and the Solr load balancer. Finally, port 3000 is mapped for “rails_1" and “rails_2” containers on the host machine to ports 3001 and 3002, respectively.

```
rails_1:
  build: rails
  links:
    - db
    - solr_lb:solr
  ports:
    - 3001:3000

rails_2:
  build: rails
  links:
    - db
    - solr_lb:solr
  ports:
    - 3002:3000
```

Listing 20. Data-only container configuration for Solr instances in YAML file

The following part of the YAML file configuration is intended to set up HAProxy load balancer for distributing HTTP requests to the Rails web application instances. The configuration is listed in Listing 21. The HAProxy load balancer container uses custom image, which Dockerfile is located in “rails/haproxy” directory under project root directory. It has name “rails_lb”, two links to both Ruby on Rails containers and exposes port 80 on the host machine.

```
rails_lb:
  build: rails/haproxy
  links:
    - rails_1
    - rails_2
  ports:
    - 80:80
```

Listing 21. HAProxy load balancer configuration for Rails instances in YAML file

The Docker Compose configuration defines the entire multi-container Trail application architecture in a single file. Furthermore, containers were customised in a convenient way using Dockerfiles and configuration files for load balancers and the web application. The complete Docker Compose configuration file is provided in Appendix 3.

4.6 Testing

This section describes the testing process of the implemented architecture solution. The purpose is to verify the implemented design is operational and meets the requirements specified in section 4.4. The testing was done on a local machine with Ubuntu Wily Werewolf 15.10 (64-bit) Linux distribution.

The first step in the testing process was to login into Ubuntu terminal under sudo privileges and locate the directory with the project. Afterwards, the application containers infrastructure was built and started up from the project directory using "`docker-compose up`" command. This command builds, creates, starts, and attaches to containers for the application. The command aggregates the output of each container and displays it in terminal [42]. When the command exits, all containers are stopped [42]. In order to run containers in the background mode and leave them running, "`docker-compose up -d`" command is used. This command will create 10 containers for the Trail application, 8 of which will be running and 2 are data-only containers intended only for data persistence. This was verified through "`docker ps -a`" command, which returns information about the currently running and stopped containers. The output of the command is shown in Figure 12.

```
raynlx@ubuntu-for-docker:~/workspace/trail$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS                                        NAMES
5cfee825e471        trail_rails_lb      "/bin/sh -c 'haproxy "   45 minutes ago      Up 45 minutes               0.0.0.0:3000->3000/tcp                       trail_rails_lb_1
f8318bf63eba        trail_rails_1       "/bin/sh -c 'rake db:"   45 minutes ago      Up 45 minutes               0.0.0.0:3001->3000/tcp                       trail_rails_1_1
80695f0666bd        trail_rails_2       "bash -c 'rails serve"   45 minutes ago      Up 45 minutes               0.0.0.0:3002->3000/tcp                       trail_rails_2_1
ce07e89f0256        trail_solr_lb       "/bin/sh -c 'haproxy "   45 minutes ago      Up 45 minutes               0.0.0.0:8080->8080/tcp                       trail_solr_lb_1
79d633f6eae3        trail_solr_2        "/bin/sh -c '/opt/sol"   45 minutes ago      Up 45 minutes               0.0.0.0:8985->8983/tcp                       trail_solr_2_1
32fab06aca73        trail_solr_1        "/bin/sh -c '(sleep 1"   45 minutes ago      Up 45 minutes               0.0.0.0:8984->8983/tcp                       trail_solr_1_1
f598897b6f1c        solr                "/bin/true"              45 minutes ago      Exited (0) 45 minutes ago                                                trail_solr_data_1
eecf8142addb        jplock/zookeeper    "/opt/zookeeper/bin/z"   45 minutes ago      Up 45 minutes               2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp   trail_zookeeper_1
e272542725c5        mysql:5.6           "/entrypoint.sh mysql"   48 minutes ago      Up 48 minutes               0.0.0.0:3306->3306/tcp                       trail_db_1
7989e773bcfe        mysql:5.6           "/bin/true"              48 minutes ago      Exited (0) 48 minutes ago                                                trail_db_data_1
```

Figure 12. "`docker-ps -a`" command output

The following step was to verify SolrCloud service via HAProxy load balancer. Currently both Solr nodes have mapped port 8983 to port 8080 on the load balancer. The last one has also mapped port 8080 on the host machine, so that the user can open the web browser and navigate by url "`http://localhost:8080/solr`" to access Solr service. In order to check available SolrCloud nodes user has to navigate by url "`http://localhost:8080/solr/#/~cloud`" as demonstrated in Figure 13. In this

case, SolrCloud has two active Solr nodes what corresponds to the initial architecture design.

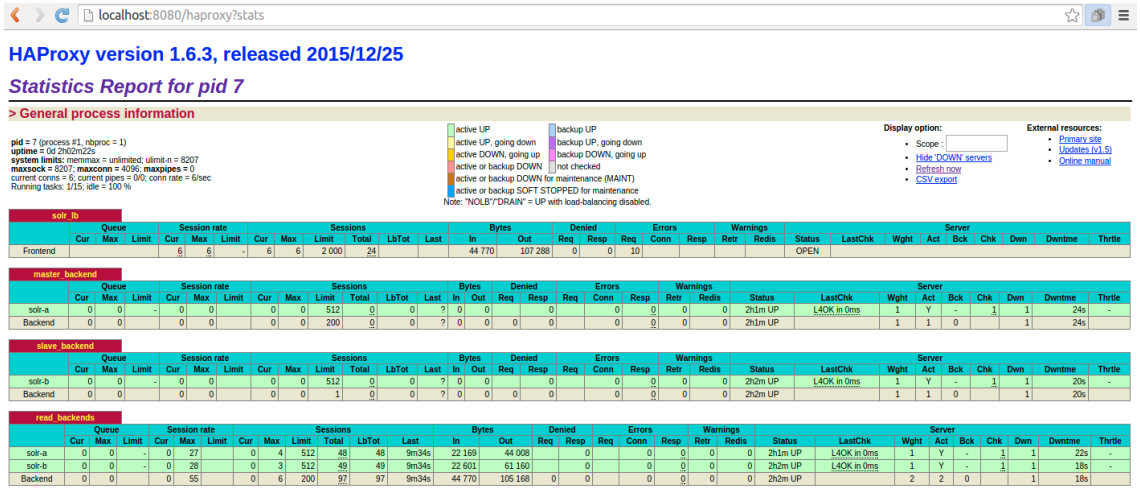

Figure 13. Statistics web page of the HAProxy load balancer for Solr nodes

The next step was to verify that HAProxy is properly configured to perform load balancing between Solr search engine instances. For this purpose, HAProxy configuration file was modified in order to enable the load balancer statistics page, which can be viewed with a web browser by url "`http://localhost:8080/haproxy?stats`". Figure 14 shows the statistics page for HAProxy load balancer used to provide high availability of the Solr nodes.

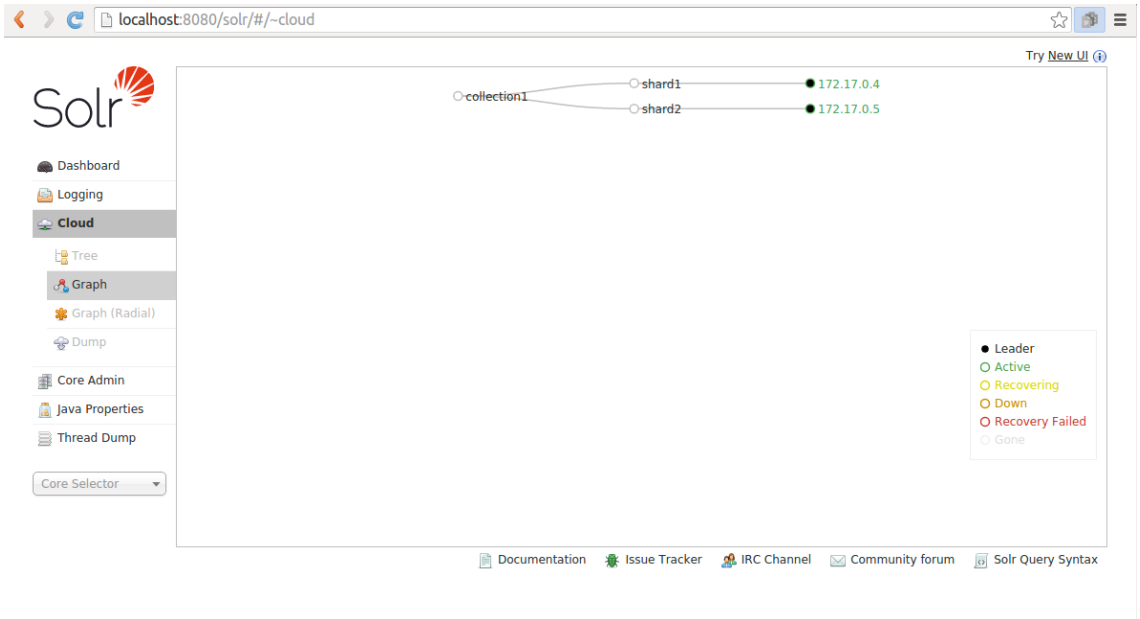

Figure 14. SolrCloud web page

The current architecture implementation has a second load balancer intended to distribute the load between Ruby on Rails web application instances. Hence, this compo-

nent also needs to be checked in order to verify its operability. Therefore, the statistics page for the HAProxy server was enabled to check the high availability of the Ruby on Rails web application instances (Figure 15).

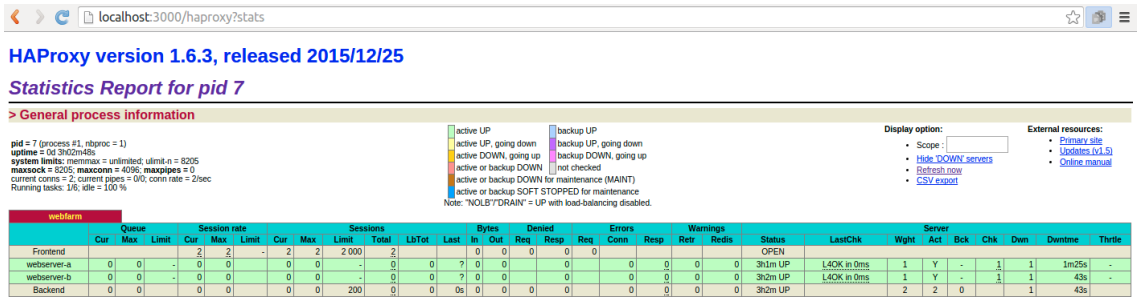

Figure 15. Statistics web page of the HAProxy load balancer for Ruby on Rails

Finally, in order to verify the service architecture and the entire Trail containers infrastructure operability on all layers, surface acceptance testing was conducted for the application. The actual Trail web application can be accessed through the HAProxy load balancer, which has mapped port 3000 on each of the Rails web application nodes with port 3000 on the load balancer. In addition, it also has mapped port 3000 and on the host machine, so that the user can open the Trail web interface through a web browser by url “`http://localhost:3000`". Hence, architecture operability was verified using the Trail web interface login page as depicted in Figure 16.

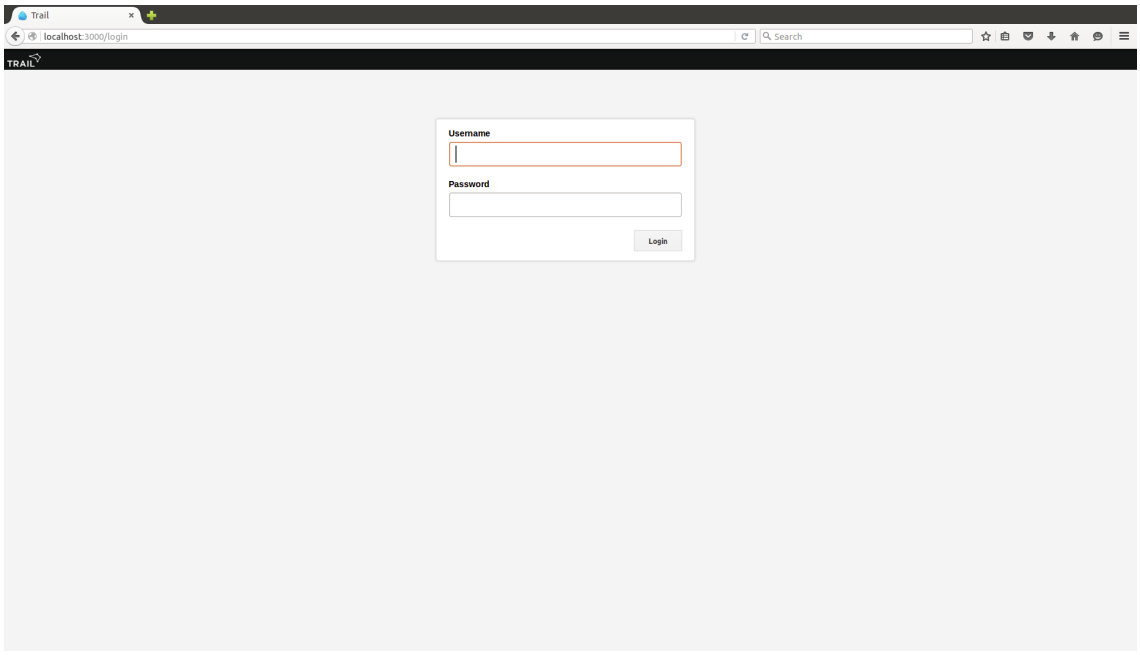

Figure 16.Trail web interface login page

The MySQL database server and Solr search engine facilities were also verified using the acceptance testing. For this purpose, multiple features of the Trail web application

were tested, requiring interaction with those services. For instance, checking Trail home page operation is demonstrated in Figure 17.

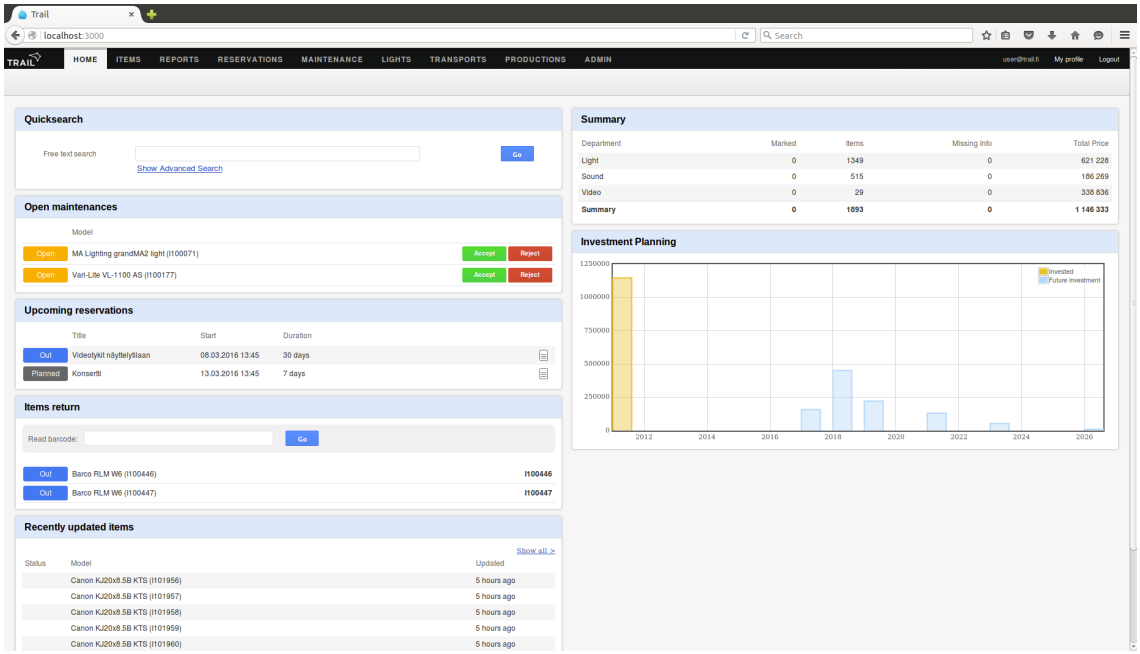

Figure 17. Trail web interface main page

To summarise, the testing phase was successfully completed. It was verified that the implemented service architecture proved to be operable and working as designed. Chapter 5 will analyse the results achieved during the development and verified through the testing.

5 Results and Discussion

This chapter summarises the project results and compares them to the objectives set in the Introduction. The goal is to verify and validate that the final outcome meets the specified requirements and provides a solution for the identified and stated problem. For this purpose, also the chosen methods of evaluating the developed service architecture for the existing product will be discussed below. Evaluation is conducted in terms of portability, scalability and usability.

5.1 Summary of Results

The project development was divided into four main phases: requirements analysis, design, implementation and testing. Each of these phases was successfully completed and described in chapter 4. During the requirements analysis phase the software requirements were gathered from the Trail Systems developers, analysed and briefly documented as described in section 4.4. The output of the design phase is the detailed specification of the software solution that accomplishes the high-level requirements defined in the previous phase. During the implementation stage, the architecture solution was built for Trail using the chosen tools and technologies in according to the detailed specification from the previous phase. Finally, testing was done in order to verify that the solution met the initial requirements.

The main outcome of this thesis is the implemented service architecture for the existing SaaS product Trail in VPC using Linux containers that meets the initial requirements for the application, listed in section 4.3:

- Application design is based on Linux containers.
- Docker and Docker Compose are used for containerisation and orchestration.
- The implemented solution is able to run in different environments like developer's local host or the production environment in VPC.

Furthermore, the project has non-functional requirements in terms portability, scalability, availability, usability and reliability. These requirements were also described in section 4.3.

The architecture implementation is a set of Docker, Docker Compose and other build and configuration files that defines Trail software components using multi-container application design. These files are used to completely automate the application setup including its dependencies. As a result, the application can be built and run within a self-contained execution environment using lightweight container-based virtualisation.

The service architecture is implemented as a multi-container application infrastructure defined within a single Docker Compose YAML file. The file contains configuration for all components and services of the application listed in section 4.4. The configuration file is used in conjunction with a set of custom Docker images listed in section 4.6. As a result, the project contains Dockerfiles to build these images. In addition, the project also contains configuration files for Ruby on Rails web application and HAProxy load balancer instances used by Dockefiles on build stage. The last part of the solution is concise documentation in the "`README.md`" file located in the project root directory. The file provides instructions for how to clone, build and run the entire application architecture. To summarise, the final solution completely matches the designed structure that meets all the technical and non-functional requirements in the project.

5.2 Evaluation of Results

The primary target of this thesis was to design and implement a multi-container service architecture for the existing product Trail making it suitable for Virtual Private Cloud. The service operation was tested and proven to work as designed. Applying the implemented Trail architecture in practice automates the application environment setup and maintenance. As a result, using Linux containers with tools like Docker and Docker Compose improves the application delivery process within a cloud.

Usage of implemented service architecture improved application delivery in multiple ways. Firstly, this approach speeds up the Trail application deployment process. Secondly introducing the new deployment method into the development environment allowed the team to focus on creating new features, fixing issues and delivering software. As a result, the solution simplifies the following IT operations:

- Easier deployment - implemented architectural solution allows deploying the application in different environments like local machine, virtual machine, private infrastructure or public cloud providers.
- Faster provisioning - built and containerised Trail application architecture can boot and restart in seconds, compared to minutes for virtual machines and days for bare metal as demonstrated in Figure 18.
- Improved resource utilisation - the primary technology for architectural solution is based on Linux containers, allowing to run all the components on a single computing instance.
- Elimination of environment inconsistencies - implemented solution automates packaging Trail with its dependencies and configurations, making the applica-

tion run in container to work as designed locally, in another environment like test or production.

- Faster introduction for new developers - the solution prevents developers spending hours to setup development environments, spinning up new instances and making copies of production code to run locally.

To summarise, the benefits of using the implemented solution listed above are proofs of portability, scalability and usability requirements.

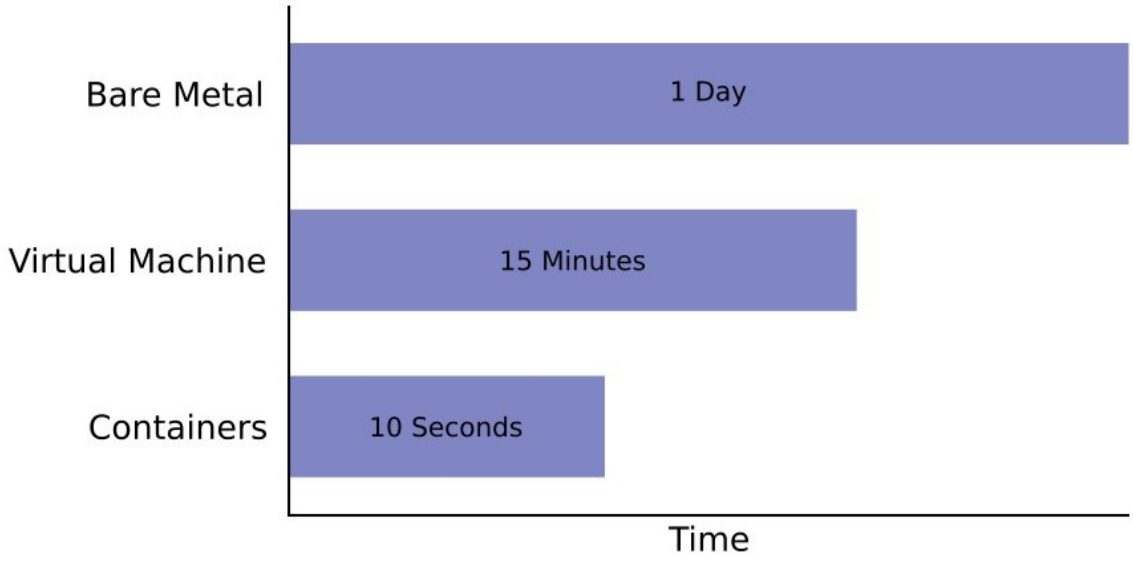

Figure 18. Estimated application delivery time for containers, VMs and bare metal

The final solution was successfully designed, implemented and tested. The results match the expected in terms of technical and non-function requirements. In this way, the main aim of the project was successfully achieved.

5.3 Development Challenges

The project was successfully completed with a few problems which were not anticipated in the design phase. On the other hand, the problems were detected and resolved in the implementation phase. This section describes the encountered issues and how they were solved.

The major challenge during the implementation process was to set up the MySQL database server for the new architecture. The reason of the problem is an incompatibility of Ruby on Rails web application framework version 3.2 with MySQL database server version 5.7. Problem was caused by the fact that MySQL version 5.7.3 m13 does not allow “`DEFAULT NULL`” value for primary key required by Ruby on Rails 3.2.

The main difficulty of the problem was to identify it. The most optimal solution in this case was to replace MySQL database server version 5.7 with version 5.6.

Another important issue during the implementation was to set up and run the distributed Solr configuration with two Solr nodes in separate containers, sharing a single ZooKeeper server instance. Due to this issue, the Trail web applicant was not able to write and read data from the Solr search engines. The problem was caused by missing configuration for creating a new SolrCloud collection. Hence, adding this command to Dockerfile of the master Solr node instance successfully solved this issue.

The last challenge was to adjust existing Ruby on Rails web application configuration for Solr search engine version 5.5. Due to the fact, that previous configuration of the Ruby on Rails application was designed to work with previous Solr version, it caused multiple minor issues. In order to resolve them, Ruby on Rails web application gems were updated, the Sunspot gem datetime patch was added and configured to enable search requests through the HAProxy load balancer to Solr instances.

5.4 Further Development

Despite the fact that current implementation has promising results and completely meets the initial requirements, there is room for further development of the solution. Current service architecture has multiple possible improvements that could be resolved to achieve the maximum benefit of using it in development and production environments. This section briefly discusses what can be improved to achieve state-of-the-art results through further development.

Current architecture implementation has only one MySQL database server node that meets the contemporary requirements of the system. On the other, it is recommended that further research can be undertaken in order to improve high availability of the database through extending existing architecture with additional MySQL database server instances. This improvement involves two demanding challenges. Firstly, all MySQL server instances have to be configured to perform master-master replication due to the fact that load balancing includes both reading and writing operations to all the backends. The second issue is to set up a load balancer between the clients and the database cluster to prevent a single server from becoming overloaded with too many requests. As a result, this enhancement will considerably improve the database accessibility for the web application.

The implemented service architecture is based on the Docker Compose orchestration product. However, the main weakness of this tool is that it does not support remote orchestration, limiting container administration to an infrastructure on a single host. Managing enterprise application scaled to multiple servers, requires more sophisticated tools such as Google Kubernetes. These tools provide high-fidelity solution for automating deployment, operations, and scaling of containerised applications. Considering the steady growth in the number of the Trail product users, this is an important issue for future research.

Another important aspect for the solution improvement is the method of creating and initialising the database. The current approach drops and creates the target database every time when administrator sets up the implemented infrastructure through "`docker-compose up`" command. This issue is caused by the fact that Ruby on Rails web application requires connection with existing database in order to be launched on a web server. Due to this, multiple instructions were added in "`rails`" Dockerfile to create, seed, migrate the database and start the web application when the container starts. Hence, this setting is suitable for test and development environments, where resetting the database is required. On the other hand, this is not acceptable for a production environment, where the database contains sensitive and valuable information. Therefore, this issue can be included in future development.

6 Conclusions

The goal of this thesis was to design and create a sustainable and reliable service architecture of an existing SaaS product, Trail, in parallel with adapting it for VPC using modern virtualisation technologies. Due to the fact that Linux containers is a new industry standard to provide modern digital services, the particular interest of this project was to implement a solution based on Linux containers using Docker and Docker Composer tools. The project aims to improve the whole Trail application delivery process enhancing the way it is built, shipped and executed.

The implemented architecture was successfully designed, developed and tested according to the initial system requirements in 5 months, which was considered to be within an acceptable timeframe. The main result of the project is multi-container Trail application infrastructure defined within Docker Compose file using additional Dockerfiles and service configuration files allowing to deploy such complex application through a single command. The architecture solution operability was tested and proven to work as designed.

The final solution improves Trail application delivery within VPC automating setup and maintenance of the application environment with its dependencies. As a result, using it in practice speeds up the Trail application deployment and provisioning. In addition, this also simplifies other routine IT tasks like resource utilisation, solving environment inconsistencies and developers onboarding. To summarise, during the project development the main features, advantages and challenges of container-based virtualisation were investigated, discussed and applied in practice.

The primary target of the project was successfully achieved in compliance with all the technical and non-function requirements determined in chapter 1. On the other hand, the solution has multiple improvements that have to be implemented in the future. Firstly, a further study with more focus on extending the existing structure with multiple instances of the MySQL database server is suggested. Secondly, further research should be undertaken in order to introduce remote orchestration and automatisation of the Trail application deployment on multiple hosts as the scope of the thesis did not cover it. Finally, further investigation is required to establish better methods for creating and initialising the database.

References

1. Trail | Friendly Asset Management. 2016. Trail | Friendly Asset Management. [ONLINE] Available at: http://www.trailassetmanagement.com/. [Accessed 04 April 2016].

2. Anastasius Moumtzoglou, 2014. Cloud Computing Applications for Quality Health Care Delivery (Advances in Healthcare Information Systems and Administration). 1 Edition. IGI Global.

3. Yohan Wadia, 2016. AWS Administration - The Definitive Guide. Edition. Packt Publishing.

4. Jason Williams, 2005. Expanding Choice: Moving to Linux and Open Source with Novell Open Enterprise Server. 1 Edition. Novell Press.

5. Oracle. 2015. Oracle® Linux Administrator's Solutions Guide for Release 6. [ONLINE] Available at: https://docs.oracle.com/cd/E37670_01/E37355/html/. [Accessed 29 November 2015].

6. Scott Gallagher, 2015. Mastering Docker. 1 Edition. Packt Publishing

7. Docker Docs. 2015. Docker Docs. [ONLINE] Available at: https://docs.docker.com. [Accessed 20 December 2015].

8. A. Srinivasan, 2014. Cloud Computing: A Practical Approach for Learning and Implementation. 1 Edition. Pearson.

9. Dr. Kris Jamsa, 2012. Cloud Computing: SaaS, PaaS, IaaS, Virtualization, Business Models, Mobile, Security and More. 1 Edition. Jones & Bartlett Learning.

10. Jose Ugia Gonzalez, 2015. Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects. 2015 Edition. Apress.

11. Brian Beach, 2014. Pro PowerShell for Amazon Web Services: DevOps for the AWS Cloud. 2014 Edition. Apress.

12. Information Resources Management Association, 2014. Cloud Technology: Concepts, Methodologies, Tools, and Applications. 1 Edition. IGI Global.

13. Thomas Erl, 2013. Cloud Computing: Concepts, Technology & Architecture (The Prentice Hall Service Technology Series from Thomas Erl). 1 Edition. Prentice Hall.

14. Abdul Salam, 2015. Deploying and Managing a Cloud Infrastructure: Real-World Skills for the CompTIA Cloud+ Certification and Beyond: Exam CV0-001. 1 Edition. Sybex.

15. Mell P, Grance T. NIST SP 800-145, A NIST definition of cloud computing [ONLINE] Available at: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. [Accessed 24 January 2016].

16. Thomas Erl, 2015. Cloud Computing Design Patterns (The Prentice Hall Service Technology Series from Thomas Erl). 1 Edition. Prentice Hall.

17. Michael J. Kavis, 2014. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). 1 Edition. Wiley.

18. Bob Familiar, 2015. Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions. 1st ed. 2015 Edition. Apress.

19. Scott Stawski, 2015. Inflection Point: How the Convergence of Cloud, Mobility, Apps, and Data Will Shape the Future of Business. 1 Edition. Pearson FT Press.

20. James Bond, 2015. The Enterprise Cloud: Best Practices for Transforming Legacy IT. 1 Edition. O'Reilly Media.

21. Lee Newcombe, 2012. Securing Cloud Services: A pragmatic guide to security architecture in the Cloud. Edition. IT Governance Publishing.

22. Rami Rosen. 2015. Linux Containers and the Future Cloud | Linux Journal. [ONLINE] Available at: http://www.linuxjournal.com/content/linux-containers-and-future-cloud. [Accessed 29 November 2015].

23. Joe Johnston, 2015. Docker in Production: Lessons from the Trenches. 1 Edition. Bleeding Edge Press.

24. Matthew Portnoy, 2012. Virtualization Essentials. 1 Edition. Sybex.

25. Navaid Shamsee, 2015. CCNA Data Center DCICT 640-916 Official Cert Guide (Certification Guide). 1 Edition. Cisco Press.

26. Fabio Alessandro Locati, 2015. OpenStack Cloud Security. 1 Edition. Packt Publishing.

27. SANS Institute Reading Room. 2015. Securing Linux Containers. [ONLINE] Available at: https://www.sans.org/reading-room/whitepapers/linux/securing-linux-containers-36142 [Accessed 29 November 2015].

28. Cisco, Red Hat. 2014. Linux Containers: Why They’re in Your Future and What Has to Happen First. [ONLINE] Available at: http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf. [Accessed 29 November 2015].

29. James Turnbull, 2014. The Docker Book.

30. Kenneth Hess, 2009. Practical Virtualization Solutions: Virtualization from the Trenches. 1 Edition. Prentice Hall.

31. Wolfgang Mauerer, 2008. Professional Linux Kernel Architecture. 1 Edition. Wrox.

32. Rami Rosen, 2013. Linux Kernel Networking: Implementation and Theory (Expert's Voice in Open Source). 1 Edition. Apress.

33. Adrian Mouat, 2015. Using Docker: Developing and Deploying Software with Containers. 1 Edition. O'Reilly Media.

34. Pethuru Raj, 2015. Learning Docker. 1 Edition. Packt Publishing.

35. Shrikrishna Holla, 2015. Orchestrating Docker. Edition. Packt Publishing.

36. Adrian Mouat, 2015. Using Docker: Developing and Deploying Software with Containers. 1 Edition. O'Reilly Media.

37. Ruby on Rails. 2016. Ruby on Rails. [ONLINE] Available at: http://rubyonrails.org/. [Accessed 31 January 2016].

38. Charles Bell, 2012. Expert MySQL (Expert's Voice in Databases). 2 Edition. Apress.

39. MySQL. 2016. MySQL. [ONLINE] Available at: https://www.mysql.com/. [Accessed 31 January 2016].

40. Sachin Handiekar, 2015. Apache Solr for Indexing Data. 1 Edition. Packt Publishing.

41. Apache Solr. 2016. Apache Solr. [ONLINE] Available at: http://lucene.apache.org/solr/. [Accessed 31 January 2016].

42. Docker Hub. 2016. Docker Hub. [ONLINE] Available at: https://hub.docker.com/. [Accessed 31 January 2016].

43. Ruby Programming Language. 2016. Ruby Programming Language. [ONLINE] Available at: https://www.ruby-lang.org/en/. [Accessed 04 April 2016].

44. HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. 2016. HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. [ONLINE] Available at: http://www.haproxy.org/. [Accessed 04 April 2016].

45. Apache ZooKeeper - Home. 2016. Apache ZooKeeper - Home. [ONLINE] Available at: https://zookeeper.apache.org/. [Accessed 04 April 2016].

46. Christopher Negus, 2013. Ubuntu Linux Toolbox: 1000+ Commands for Ubuntu and Debian Power Users. 2 Edition. Wiley.

Appendix 1. HAProxy load balancer configuration file for Solr nodes

```
global
    log 127.0.0.1   local0
    log 127.0.0.1   local1 notice
    maxconn 4096

defaults
    log     global
    mode    http
    retries 3
    maxconn 2000
    balance roundrobin
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
    stats enable
    stats uri /haproxy?stats

frontend solr_lb
    bind 0.0.0.0:8080
    acl master_methods method POST DELETE PUT
    use_backend master_backend if master_methods
    default_backend read_backends

backend master_backend
    server solr-a solr_1:8983 weight 1 maxconn 512 check

backend slave_backend
    server solr-b solr_2:8983 weight 1 maxconn 512 check

backend read_backends
    server solr-a solr_1:8983 weight 1 maxconn 512 check
    server solr-b solr_2:8983 weight 1 maxconn 512 check
```

Appendix 2. HAProxy load balancer configuration file for Rails nodes

```
global
    log 127.0.0.1  local0
    log 127.0.0.1  local1 notice
    maxconn  4096

defaults
    log      global
    mode     http
    option   httplog
    option   dontlognull
    retries  3
    option   redispatch
    maxconn  2000
    timeout  connect 5000ms
    timeout  client  50000ms
    timeout  server  50000ms

listen webfarm
    bind     0.0.0.0:3000
    mode     http
    stats    enable
    stats    uri /haproxy?stats
    balance  roundrobin
    option   httpclose
    option   forwardfor
    server   webserver-a rails_1:3000 check
    server   webserver-b rails_2:3000 check
```

Appendix 3. Docker Compose YAML file

```
db:
  image: mysql:5.6
  volumes_from:
    - db_data
  environment:
    - MYSQL_ROOT_PASSWORD=password
  ports:
    - 3306:3306

db_data:
  image: mysql:5.6
  volumes:
    - /var/lib/mysql
  entrypoint: /bin/true

solr_lb:
  build: docker/solr/haproxy
  links:
    - solr_1
    - solr_2
  ports:
    - 8080:8080

zookeeper:
  image: jplock/zookeeper
  ports:
    - 2181:2181

solr_1:
  build: docker/solr/solr_1
  volumes_from:
    - solr_data
  links:
    - zookeeper
  ports:
    - 8984:8983
```

```
solr_2:
  build: docker/solr/solr_2
  volumes_from:
    - solr_data
  links:
    - zookeeper
  ports:
    - 8985:8983

solr_data:
  image: solr
  volumes:
    - /opt/solr/server/solr
  entrypoint: /bin/true

rails_lb:
  build: docker/rails/haproxy
  links:
    - rails_1
    - rails_2
  ports:
    - 3000:3000

rails_1:
  build: .
  links:
    - db
    - solr_lb:solr
  ports:
    - 3001:3000

rails_2:
  build: .
  links:
    - db
    - solr_lb:solr
  ports:
    - 3002:3000
```

```
command: bash -c "rails server"
```